AFIT/GCS/ENG/93D-02

DESIGN OF A PARALLEL DISCRETE EVENT SIMULATION COPROCESSOR

THESIS

Jacob Lanier Berlin
Captain, USA

AFIT/GCS/ENG/93D-02

AFIT/GCS/ENG/93D-02

DESIGN OF A PARALLEL DISCRETE EVENT SIMULATION COPROCESSOR

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

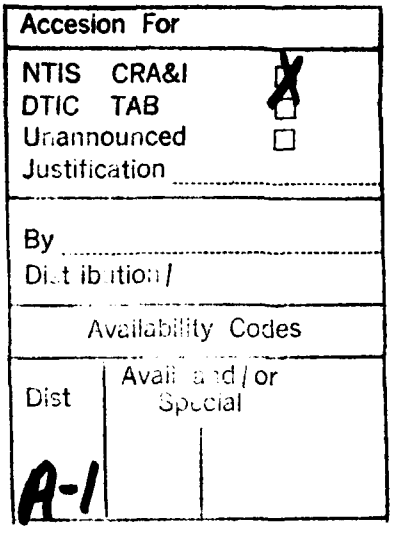
Requirements for the Degree of

Master of Science in Computer Engineering

Jacob Lanier Berlin, B.S.

Captain, USA

December, 1993



Approved for public release; distribution unlimited

Table of Contents

List of Figures

List of Tables

Abstract

A Parallel Discrete Event Simulation Coprocessor was designed to off-load the synchronization overhead from the processors executing the application. In a multiprocessor architecture, one coprocessor executes the synchronization routines for each host processor. Speedup can be achieved when the host processor executes the application and the coprocessor concurrently executes synchronization routines. The coprocessor uses a programmable microcode control store to guarantee flexibility in the synchronization routines.

The coprocessor uses an Extreme Search Associative Memory to support fast Next Event Queue (NEQ) management. This associative memory uses bit-serial word-parallel search logic to provide O(1) insert and retrieval time of events in the NEQ.

The coprocessor was completely described in the VHSIC Hardware Description Language (VHDL), and several components were fabricated and tested. Timing measurements of the fabricated components were back-annotated into the VHDL description to improve model accuracy.

Synchronization overhead of a parallel VHDL simulation was measured using the AFIT Algorithm Animation Research Facility, and this data was used for a conceptual performance analysis of the coprocessor. A four-fold speedup was achieved for the NEQ management of the simulation; however, the total speedup was only 1.02 since less than 2% of the application was accelerated.

Although the coprocessor is designed as an I/O board consisting of multiple integrated-circuit packages, this research provides proof-of-concept for on-chip synchronization hardware of future processors.

DESIGN OF A PARALLEL DISCRETE EVENT SIMULATION COPROCESSOR

I. Introduction

1.1 Background

Computer simulations play an important role in a diverse range of applications. For example, due to the high cost and complexity of Very High Speed Integrated Circuits (VHSICs), they must be thoroughly simulated prior to fabrication. Unfortunately, simulations in engineering, computer science, economics, and military applications require unacceptable amounts of time to execute on sequential machines. Consequently, these simulations have become a significant bottleneck within their application domain (8:19). For example, some simulations may require months of computer time to produce accurate results (7:449).

Because of the importance of simulations and their increasingly complex nature, speedup has become a vital issue in simulation research. The three basic techniques for simulation speedup involve technology, ingenuity, and architecture. Technology improvements consist predominantly of using a faster logic family. Ingenuity improvements include the development of more efficient algorithms and data structures. Finally, architectural improvements include hardware specialization and exploitation of concurrency through parallelism and pipelining (7:449-450).

Achieving several orders of magnitude speed increases in simulations will require the use of one or more algorithmic or architectural improvements (7:450). Furthermore, as computational requirements of simulations increase, even the fastest sequential processors cannot adequately fulfill the requirements (19:8). These realizations have provided the motivation for extensive research in the area of parallel simulations.

Parallel simulations are executed on multiprocessor architectures. Portions of the simulation are executed in parallel on different processors of the multicomputer network. The portions that are executing in parallel interact with each other by passing messages

across the network. Amdahl showed that the potential speedup from exploiting the parallelism of an application is limited by the serial fraction of the application (11:532). For example, an application with a serial fraction of fifty percent can at best receive a two-fold speedup from parallel execution.

1.2 Problem

The use of parallel architectures for simulation speedup has several inherent problems which must be overcome in order to realize a performance increase. These problems include the communications overhead between the logical processes of the simulation and the synchronization delays to insure that the simulation proceeds in the correct time order (6:2-3).

Another significant simulation problem is the overhead of event-list management. Many simulations must maintain event data in long lists. As the size of these lists grows, the amount of time required to search for and insert or remove events from the list also increases. For example, some versions of the AFIT Carwash simulation can spend over fifty percent of their execution time managing the NEQ data structure.

The objective of this research was to complete the design and implementation of a coprocessor which would reduce the communication and synchronization overhead associated with parallel simulations. The design also incorporated an associative memory to support fast list management. Although the target architecture was the Intel iPSC/2 Hypercube, this research is relevant to the design of any future processor which may have similar on-chip features to support parallel processor synchronization and simulation event-list management.

1.3 Summary of Current Knowledge

Taylor proposed that a Discrete Event Simulation (DES) Coprocessor could provide significant speedup to parallel simulations. His thesis work included a top-level design of this coprocessor and calculations which indicated that the coprocessor could provide speedup ranging from 1 to 20 depending on the number of logical processes running on

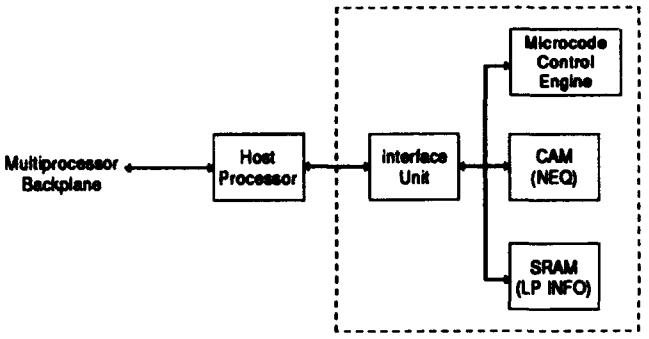


Figure 1. Simplified DES Coprocessor Architecture (DES portion in box)

a node (24). Daniel followed Taylor's thesis work and implemented the high-level design as a behavioral circuit in the VHSIC Hardware Description Language (VHDL). Daniel conducted an analysis which indicated that the coprocessor could achieve speedup ranging from 1.2 to 60 depending on the granularity of the simulation (6:86).

The four major components of the DES coprocessor are the microcode control engine, an SRAM, a Content Addressable Memory (CAM), and an interface unit (Figure 1). The microcode control engine has a programmable control store and executes all of the synchronization tasks of the simulation. The SRAM stores the simulation-specific data which is required to synchronize the simulation. The CAM was provided by Banton from his doctoral research (1:50-81). This component uses parallel-search logic to provide O(1) insert and retrieval time of events in the Next Event Queue. Finally, the interface unit transfers data between the host and the coprocessor. In a multiprocessor architecture, each DES Coprocessor supports only one host processor.

1.4 Assumptions

Significant assumptions which were made at the start of this research are listed below. Several of these assumptions were invalid and delayed the progress of this research. The impact of these failed assumptions will be discussed in greater detail in Chapter 3.

- Daniel's behavioral description of the coprocessor was functionally correct.
- Banton's Extreme Search Associative Memory (ESAM) was ready for fabrication and could be incorporated into the DES Coprocessor without modification to the ESAM architecture.
- The documentation on the current design was sufficient and correct.
- Automated Computer-Aided Design (CAD) tools could successfully generate transistor layouts and require a minimum of full-custom design.
- Intel would release sufficient proprietary information on the interface requirements for the iPSC/2 Hypercube. The physical interface, logic design, and timing requirements were required so that the coprocessor could be properly interfaced.
- All fabricated circuits would be in a 2.0 μm process.

1.5 Scope

This research effort was limited to completing the coprocessor design of Taylor and Daniel (24, 6). Daniel's VHDL behavioral description of the coprocessor was partially implemented as a physical circuit using a mixture of custom designed devices, commercial programmable logic devices, and commercial memory devices. For ease of implementation, the design was targeted as an I/O board which could be wire-wrapped. A performance analysis was conducted based on data extracted from the partially completed design.

Although the DES coprocessor was designed to support a wide range of simulation paradigms, only the Chandy-Misra approach was implemented and analyzed. A parallel VHDL simulation taken from Breeden's research was used as performance benchmark (2).

1.6 Approach

A four step approach was taken in this research:

- Analysis of Baseline Design - The first step was to analyze Daniel's DES Coprocessor design. This step included a review of the design documentation, the VHDL source code, the test cases to validate performance, and the speedup predictions.
- Requirements Analysis - The next step was to determine the procedures that would be necessary in order to successfully complete this research. Since research assumptions were proven to be invalid, it was necessary to modify both the requirements and the scope of this research.
- Implementation - This step consisted of redesigning the coprocessor architecture in accordance with the requirements analysis. This step also consisted of decomposing the completed VHDL description of the coprocessor into a viable physical circuit. Field Programmable Gate Arrays, commercial memories, and custom-fabricated circuits were necessary to complete the design. The structurally decomposed circuit was implemented only partially due to time and resource constraints.
- Testing and Performance Analysis - The fabricated components were tested for functionality and parametric performance. A performance analysis of the coprocessor was conducted based on data collected from implemented components and CAD simulation.

1.7 Overview

The remainder of this document is divided into five sections: Literature Review, Methodology, Implementation, Findings, and Conclusions. The Literature Review takes a brief look at current Parallel Discrete Event Simulation algorithms and hardware support. The Methodology chapter explains the analysis of Daniel's coprocessor design and the requirements analysis. The Implementation chapter explains the finalized design of the coprocessor and important design decisions that were made to arrive at that design. The Findings chapter explains the testing and performance analysis of fabricated components and the top-level coprocessor design. The Conclusions chapter discusses the impact of the

findings and makes recommendations for follow-on research. The appendices elaborate on technical design information and appear in the order that they are referenced.

II. Literature Review

2.1 Introduction

A broad range of issues must be considered in the design of a parallel discrete event simulation hardware accelerator. For the accelerator to be general purpose, it must support the spectrum of algorithms which implement parallel discrete event simulations. Furthermore, current hardware accelerator designs for parallel discrete event simulations should be considered to determine potential pitfalls within the proposed design.

This chapter provides a brief description of parallel discrete event simulation followed by a review of the algorithms which support this type of simulation. The final portion of the chapter covers some current hardware accelerator designs for parallel discrete event simulation.

2.2 Parallel Discrete Event Simulation

Discrete event simulations are used across a broad domain of applications including engineering, economics, computer science, and military planning. These applications often require large simulations which use an unacceptable amount of time to execute on sequential machines (7:449). Parallel Discrete Event Simulation (PDES) is being researched as a viable technique to accelerate these large applications.

2.2.1 PDES Explained. A discrete event simulation is a program that models objects which change state only at discrete points in simulated time. A change in state is associated with the occurrence of an event at the object being simulated. The simulation time advances and an object's state changes only when events occur (8:19). Simulation time does not progress continuously but "jumps" based on the occurrence of events.

A discrete event simulation can be executed in parallel by decomposing the system to be simulated into Logical Processes (LPs). Each LP represents a Physical Process (PP) which is an actual component within the system being simulated. A CPU inside a computer is an example of a physical process within a physical system (the computer). A description language model of a CPU is an example of a logical process within a logical system (the

computer model) (17:54). A physical system can be simulated as a set of physical processes that operate autonomously and interact with each other by passing messages. In general, an event at one physical process that causes events at other physical processes can be simulated by message communication (17:54).

To implement PDES, the LPs are mapped to different processors and execute sequential code in parallel. Communication arcs are established between LPs which interact. Each LP can send and receive messages on its communication arcs. An event at one LP that causes events at other LPs is simulated by messages that are passed on the communication arcs (17:42-45). Each LP maintains its own event list and simulation time. Since the distributed LPs are executing asynchronously and in parallel across multiple processors, it is unlikely that LPs will have the same current simulation time.

2.2.2 PDES Synchronization. Valid simulations must process calculations in a correct time order. These calculations are distributed across all the LPs and processors being used. The correct order of these calculations is data dependent and not known until runtime (8:19-20). PDES algorithms use synchronization mechanisms to insure that calculations are executed in the correct order while achieving optimal performance.

2.3 PDES Algorithms

PDES algorithms have typically been categorized into two approaches: conservative and optimistic. The basic premise of the conservative approach is that no LP within the simulation will receive a message out of the past (i.e., with a lower time stamp than the local simulation time of the LP). The basic premise of the optimistic approach is that any LP within the simulation can restore its state to a time prior to any time-stamped message that is received. Reynolds defined these rules as the endpoints of an entire spectrum of possibilities for parallel discrete event simulation algorithms (20:325-327). The Chandy-Misra approach is the most notable conservative approach, and Jefferson's Virtual Time approach is the most notable optimistic approach.

2.3.1 The Chandy-Misra Approach. The Chandy-Misra approach uses null messages to prevent LPs from receiving out of order time-stamped messages and to prevent deadlock of the simulation. Deadlock can occur if an LP is waiting to receive a message from another LP which does not have a message to send. This problem can occur in both acyclic and cyclic networks (17:55-56).

A real message that is sent by LP_i to LP_j indicates that an event has occurred at LP_i which may affect the state of LP_j. In contrast, a null message that is sent by LP_i to LP_j indicates that LP_i will not send another message to LP_j any sooner than the time specified in the null message. There is no equivalent to null messages in a physical system, so this method introduces simulation overhead (4:201-202). Null messages are sent from one LP to another to indicate that it is safe for the "downstream" LP to proceed to the time indicated in the message.

The basic algorithm is that an LP receives messages allowing it to advance in time. The LP can then alter its internal state, update its local clock, and generate real messages for the appropriate outgoing arcs. Then the LP generates and sends null messages for the arcs which did not receive real messages. The fundamental principle of this algorithm is that an LP will never receive a message out of the past (17:57).

2.3.2 Performance of the Chandy-Misra Approach. The use of null messages to synchronize computations and avoid deadlock introduces a significant overhead. For example, the number of null messages required to synchronize a simulation is directly proportional to the number of LP communication arcs (16:128). A large proportion of null messages will cause a bottleneck in the communication network and slow the simulation down. Reed determined that significant parallelism is required in order to regain the losses from the null message overhead. The communication overhead can only be amortized when the LPs interact infrequently, which is a rare case (19:11).

Another shortcoming of the Chandy-Misra approach is that LPs must block until they are guaranteed that it is safe to proceed. This inactivity in the simulation can be viewed as another form of overhead in the protocol (15:86).

Other major shortcomings of the Chandy-Misra approach are that it does not fully exploit parallelism, it requires static configurations, and it usually requires the programmer to be involved with the synchronization mechanism (8:24). Because of these limitations, the Chandy-Misra approach would be optimal for a simulation that can be easily decomposed into a static network with regular message traffic (15:84-86).

2.3.3 The Virtual Time Approach. Jefferson proposed the virtual time paradigm and the Time Warp mechanism used to implement that paradigm (13). Lookahead-rollback is the fundamental synchronization feature of this paradigm. Each LP is allowed to run as fast as possible without regard to synchronization conflicts with other LPs. When a conflict occurs (for example, a message is received out of the past), the offending LPs are rolled back in time to a point prior to the conflicting time. Jefferson justifies this optimistic approach based on the following arguments (13:405):

- Distributed rollback can be implemented easily and efficiently.
- Other conservative methods would have simply blocked for a time equal to the amount of wasted computation when rollback occurs.
- Rollback should occur infrequently.

Local Virtual Time (LVT) is defined as the simulation time of a particular LP. This time generally progresses forward, but will jump backward when a rollback operation occurs. Global Virtual Time (GVT) is defined as the minimum of all the LVTs within the system. While LVT may occasionally go backwards, GVT always progresses forward. Irrevocable operations such as I/O are executed according to the GVT (13:410-412).

Rollback is accomplished through the use of state queues and anti-messages. The state queue is used to save the current state of the process. When a process needs to roll back, the old state is recovered from the queue. The frequency of state saving can impact on the efficiency of the simulation. In general, the frequency of state saving can be based on the amount of elapsed time (simulation or wall-clock) or the number of events occurring (13:412).

Anti-messages are used to cancel messages that must be undone whenever there is a rollback. These messages are created at the same time as the real messages and are held in an output queue in case they are needed. When an LP must roll back, it sends the anti-messages corresponding to the real messages that have already been sent. If the anti-message is received while the actual message is still in the receiver's queue, the two messages annihilate each other and the receiver does not have to roll back. If an anti-message is received after the original message has already been executed, then the LP must roll back and send anti-messages to undo any messages it may have sent (13:412-416).

2.3.4 Performance of the Virtual Time Approach. The major shortcomings of Jefferson's approach are the state saving overhead, the excessive use of memory, and an increase in design complexity (8:26). The state saving overhead is incurred whenever an LP must archive a copy of its state to a protected memory location. State saving may be required as frequently as the occurrence of every event at the LP. This state saving requires time and also consumes excessive amounts of memory. It has been shown that optimistic algorithms in general require several times as much memory as conservative approaches (8:26).

Because of these shortcomings, Jefferson introduced the Cancelback Protocol as an extension of the Time Warp Operating system. This protocol reduces the state saving overhead by limiting the amount of lookahead that is allowed (14). Although this protocol guarantees that the simulation will require no more memory than a sequential algorithm, it does so at the expense of time performance. As the degree of space is optimized, the simulation behavior reduces to a sequential algorithm with the event list spread over many processors (14:2).

Based on its strengths and weaknesses, Jefferson's approach would work best in a simulation in which numerous processes communicate infrequently with each other or in a network that changes dynamically (15:84-86).

2.3.5 A Spectrum of Options. Reynolds claims that the dichotomy of PDES into conservative or optimistic approaches is incorrect. There exists a spectrum of options for

PDES implementations, and the conservative and optimistic approaches are only a subset of the entire spectrum (20). He gives a set of design variables which define the design space of PDES approaches. These design variables are (20:327-329):

- Partitioning - LPs may be partitioned into clusters that employ different synchronization strategies.
- Adaptability - LPs may dynamically change design variables such as the amount of lookahead allowed.
- Aggressiveness - LPs may process messages based only on conditional knowledge.
- Accuracy - The requirement that a simulation ultimately process events in the correct sequence.
- Risk - LPs may pass messages which have been processed based only on conditional knowledge.
- Knowledge Embedding - LPs may have knowledge of other LPs behavioral attributes.
- Knowledge Acquisition - LPs may request knowledge from other LPs
- Synchrony - The degree of temporal binding among LPs.

Based on these design variables, there is essentially an infinite design space for PDES in which some alternatives will be optimal for particular applications.

2.3.6 SPECTRUM Testbed. The SPECTRUM (Simulation Protocol Evaluation and Concurrent Testbed with ReUsable Modules) testbed was designed to provide a common environment for testing and comparing a variety of PDES algorithms across these eight design variables (21). The testbed is composed of an application component, a process manager, and a node manager per LP. If there are multiple LPs running on one node there is only one node manager. The application component has access to several operations within the process manager. These operations are:

- Initialization, a one-time operation at simulation start-up to put each LP in an initial state.

- Post-event, an operation to send out messages from an LP.
- Get-next-event, an operation to get the next time-ordered message from the Next Event Queue.
- Time advance, an operation to update the local simulation time.

The node manager has access to a post-message operation in the process manager. This operation allows incoming messages to be placed in the Next Event Queue (NEQ). The SPECTRUM testbed was designed to support the entire range of PDES algorithms so that they could be tested and compared in a common environment. Unfortunately, Reynolds found that the dependence between protocols and applications was greater than expected. Therefore, the testbed requires some reconfiguration for different applications (21:671).

2.4 PDES Hardware Acceleration

PDES algorithms attempt to overcome inherent synchronization problems, and each algorithm has both strengths and weaknesses. To compensate for the weaknesses of these algorithms, specialized hardware has been designed. The most notable hardware designs are Fujimoto's Rollback Chip and Reynold's Parallel Reduction Network.

2.4.1 The Rollback Chip. Fujimoto designed the Rollback Chip to compensate for the overheads incurred in Jefferson's Time Warp mechanism (9). The significant overheads of the Time Warp mechanism are state saving and rollback. For example, a state save after each event may occur every 100 microseconds, and each state save may require one megabyte of memory (10:69-70).

A hardware accelerator for Time Warp must optimize both state-saving and rollback. This is a difficult task since these two features are mutually exclusive parameters for optimization (9:401).

The Rollback Chip (RBC) is a specialized Memory Management Unit and data cache combined into one component (Figure 2). The word "chip" is somewhat misleading since the design contains too much circuitry to fit in one integrated circuit (10:77). The RBC uses large stack frames (4 Mb) of state history which are implemented as a circular buffer

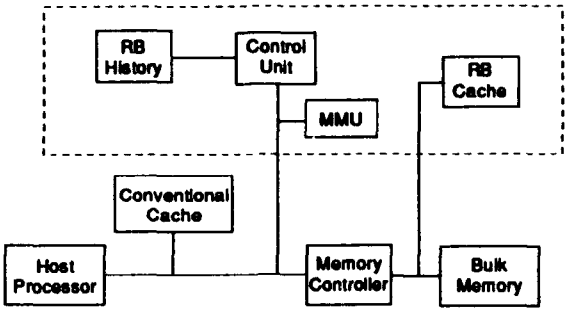


Figure 2. Simplified Rollback Chip Architecture (RBC portion in box)

in memory. While the stack is maintained in bulk memory, the RBC also has a high speed data cache which maintains the most recent version of each data line. The RBC also has a buffer unit (RB History) which maintains the top of the stack and allows faster rollback updates.

Fujimoto conducted a conceptual performance analysis of the Rollback Chip. He compared a two node PDES without RBC to a two node PDES with RBC (3). The performance of the non-RBC implementation was affected by several factors. The state copying required by the simulation could approach ninety percent of the total execution time. The memory usage was inefficient and could quickly consume all of the available memory. The operating system overhead of memory allocation/deallocation and garbage collection hindered the simulation performance. On the other hand, the cost of rollback was inexpensive because a rollback simply consisted of changing a memory pointer to the correct state in memory (3:153-154).

Fujimoto also examined the performance of the RBC implementation. The overhead of state copying was completely eliminated since the RBC did this transparently. The memory usage was reduced since the RBC only copied data that was modified. The memory management was handled entirely by the RBC which reduced the operating system

overhead. Finally, the rollback overhead increased slightly since the RBC had to do extra work to find the location of the most recent data.

Fine grain simulations showed the most potential for speedup (from 1.21 to 62.4) while coarse grain simulations had the worst potential for speedup (from 1 to 2.6). The large variance in speedup was due mostly to the state sizes and frequency of message passing. Simulations with larger state sizes and more message passing had better speedup (3:155-156).

The Rollback Chip is an example of decentralized hardware accelerator for the Virtual Time paradigm. Like the RBC, the AFIT DES Coprocessor is also a decentralized accelerator. Furthermore, the RBC design provides insight as to how the DES Coprocessor could be configured to support optimistic PDES algorithms.

2.4.2 Parallel Reduction Network. Reynolds proposes a Parallel Reduction Network (PRN) to support the rapid dissemination of global synchronization information for PDES (22). The PRN consists of a pipelined binary tree of ALUs which execute associative operations on data provided from each host processor (Figure 3). The PRN is capable of providing global information very rapidly and is capable of supporting both conservative and optimistic approaches (22). In support of conservative approaches, the PRN can rapidly disseminate minimum look-ahead values to each host processor. In support of optimistic approaches, the PRN can rapidly disseminate the GVT.

The primary objective of the PRN design was to separate the processors that execute the application from those that synchronize the application. Other design goals were to make the PRN architecture:

- Transparent to the application.
- Capable of retro-fitting to existing architectures.
- Cost much less than the cost of the host architecture.
- Independent of the host architecture network.

Reynolds predicts that his design will be able to calculate and disseminate global information across a 32 processor machine in 750 nanoseconds (22:2). Although this pre-

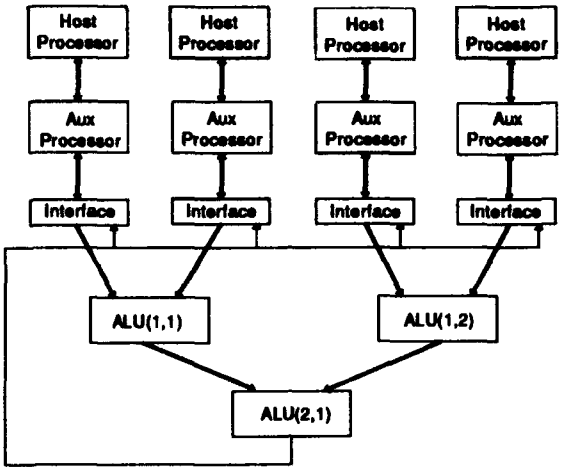


Figure 3. Simplified Four Node Parallel Reduction Network

diction implies potential speedup in simulations, Reynolds has yet to do a comparative performance analysis between his proposed design and a typical parallel architecture.

The PRN is an example of a centralized general purpose PDES hardware accelerator which is based on the dissemination of global information. Although the PRN is different from the AFIT DES Coprocessor in terms of architecture, both designs attempt to achieve general purpose PDES acceleration. The PRN implementation provides some insight into different ways the DES Coprocessor can be configured as a general purpose accelerator. Furthermore, the design goals of the PRN also serve as a good foundation of design goals for the DES coprocessor.

2.5 Conclusion

This chapter presented a brief look at Parallel Discrete Event Simulation algorithms and special purpose hardware which supports these algorithms. In order for the DES Coprocessor to be general purpose, it must support the entire range of PDES algorithms. Reynolds designed the SPECTRUM testbed to support the entire range of algorithms across eight design variables; however, he found that some reconfiguration was required based on the application. Since the AFIT DES Coprocessor has a programmable control

store, it will be able to support a flexible testbed and still provide hardware acceleration of synchronization tasks. Finally, both Fujimoto's Rollback Chip and Reynold's Parallel Reduction Network were examined and compared with the AFIT DES Coprocessor.

III. Methodology

3.1 Introduction

As discussed in chapter one, a four-step approach was used in this research: analysis of baseline design, requirements analysis, implementation, and testing/performance analysis. This chapter provides a detailed explanation of the analysis of the baseline design and the requirements analysis.

3.2 Analysis of Baseline Design

The baseline design was provided by Daniel as his master's thesis and is documented in (6). The analysis of his design consisted of a review of his speedup predictions and VHDL behavioral description of the circuit.

3.2.1 Analysis of Predicted Speedup. A review of Daniel's performance analysis revealed that the speedup potential of the coprocessor had been overstated. Daniel determined that the speedup provided by the DES coprocessor for the SPECTRUM Get Event routine was 1575 while the speedup provided for the SPECTRUM Post Message routine was only 297. This is a contradictory result because the SPECTRUM Get Event routine occurs in O(1) time complexity while the SPECTRUM Post Message routine occurs in O(n) time complexity. Since the CAM provides both O(1) insert and retrieval time of events from the NEQ, the speedup of the Post Message should be greater than the speedup of the Get Event.

Based on this anomaly, a more detailed analysis was conducted with a focus on the speedup provided by the CAM implementation of the NEQ over the SPECTRUM software implementation (a prioritized, doubly-linked list). The analysis revealed that the amount of application time spent managing the NEQ was insignificant for parallel VHDL simulations and only became noticeable in Carwash simulations when the processor loads were intentionally unbalanced to create bottlenecks in event processing. For example, the average NEQ size for the Wallace Tree multiplier was four events and queue management required an unmeasurable amount of execution time. For the severely unbalanced Carwash

simulations, the average NEQ size approached eight hundred events and queue management consumed up to thirty percent of the application time.

Although this initial analysis raised doubt as to the true speedup potential of the coprocessor, the research continued based on the following realizations:

- The actual speedup would probably be less than originally predicted.
- The potential speedup could be increased by using the coprocessor for application event-list management instead of SPECTRUM event-list management.
- The potential speedup could be increased by interfacing the DES coprocessor between each host processor and the multiprocessor backplane, thus off-loading all of the message passing overhead from the host processor.

3.2.2 Analysis of VHDL Behavioral Description. An analysis of the VHDL behavioral description of the coprocessor revealed several significant problems which are discussed below.

3.2.2.1 Documentation Errors. The functionality of the coprocessor was impossible to determine because of a lack of accurate, detailed schematics. In particular, the following documentation errors were observed:

- Schematic names for components and signal lines were different than the actual names used in the VHDL behavioral description.
- Schematics reflected incorrect connectivity of components.
- Schematics did not represent all of the connectivity between components.
- The design contained behavioral components which were not shown in any schematics.

3.2.2.2 Inaccurate Component Models. Behavioral descriptions did not accurately model the components. For example, sensitivity lists of components did not include all of the appropriate signals for process activation. For a combinational circuit,

all inputs should be in the sensitivity list. This fundamental rule was frequently violated. As a result, the timing of signals in the design was haphazard and functionality was based more on chance than design. Furthermore, predefined attributes such as 'TRANSACTION were used improperly and further distorted the accuracy of the component models.

3.2.2.3 Validation of Output. The coprocessor output could only be observed through the Synopsys Interactive Waveform Viewer as binary, octal, decimal, or hexadecimal values on signal lines. The coprocessor output consists of interrupt signals/vectors, error signals/vectors, and data packets. Validation of this quantity of output using the Waveform Viewer was very difficult and time-consuming for the designer. Furthermore, extensive Waveform Viewer output caused slower simulations due to increased I/O requirements. Validation of output by this method was prone to error.

3.2.2.4 Microcode Routines. The programmable control store of the DES coprocessor is loaded from a microcode input file. The following observations were made while reviewing this input file:

- The microcode input file was in decimal format and uncommented. This made it impossible for the programmer to read and understand the input file without extensive cross-referencing to other documentation.
- The pseudo-code documentation of the microcode contradicted the microcode input file.
- The microcode input file specified an absolute address for each microinstruction and each branch target. This made it difficult to insert, delete, or move blocks of code.
- The microcode contained undocumented microinstructions.

3.2.2.5 Timing Analysis. The propagation delays for the behavioral components were based on a four-phased non-overlapping clock running at 25 MHz. The master clock cycle was 40ns, so the non-overlapping pulses had to be less than 10ns. Wire wrapping and off-chip capacitive loading would require slower speeds.

3.3 Initial Requirements Analysis

The requirements analysis consisted of determining what needed to be done based on the results of the baseline analysis. The following procedures were determined to be necessary for the successful completion of this research.

3.3.1 Schematic Capture. The behavioral description of the coprocessor needed to be accurately represented with detailed schematics to determine circuit connectivity and functionality. The Synopsys Graphical Environment (SGE) CAD tool would be used to create schematics of the existing circuit. Schematic capture would insure that the schematics accurately represented the behavioral design. All behavioral descriptions of components from the baseline design would initially be re-used so that an accurate hierarchy of schematics which represented the baseline design could be created. All design changes would also be made with SGE so the final product would be accurately represented by schematics.

3.3.2 Validation of Coprocessor Output. Waveforms were inadequate for validating the coprocessor performance. The testbench needed to be modified to capture the coprocessor output, convert it into a readable format, and write it to a file. The Waveform Viewer would be used for debugging, and text output would be used for validation. Different design versions could then be quickly validated by using the Unix "diff" command between previously validated results and new design results.

Multiple testbenches would be used to validate performance. Simple, fast tests would be used during the initial stages of a design modification when errors were most likely. More thorough, time consuming tests would be used at the end of a design modification to catch any subtle errors in the design.

The testbench would also have to be rewritten to be more modular. Repetitive sequences of stimuli would be called from procedures, thus reducing the amount of code, making it more readable, and also making it easier to validate performance.

3.3.3 Microcode Preprocessor. A preprocessor needed to be written for the microcode input file so that the file could be commented and easier to modify. The ability to have comments in the file was essential. Due to the complexity of the code, it would be

necessary to have one comment per line of microcode. The addressing scheme of the microcode would also have to be modified. A relative addressing scheme for branching would be developed to make it easier to modify the code. Finally, the preprocessor would have to determine the absolute addresses of instructions at run-time and free the programmer of that burden.

3.3.4 ESAM Upgrade. The extreme search capability of the CAM was very cumbersome and required extensive peripheral logic in order to work properly. For example, Daniel had to design a front-end driver to do a bit-wise search of the memory to locate the minimum time-stamped next event. The CAM would be replaced by the ESAM which would do this search internally. A small finite state machine would be designed to control the peripheral logic and offload the complexity of ESAM control from the coprocessor control engine. This finite state machine would also allow for concurrency between the ESAM and coprocessor control engine, thus increasing the potential for acceleration.

Both the CAM and ESAM VHDL descriptions placed the most-significant data bit in the lowest-numbered bit. The remainder of the coprocessor description placed the most-significant data bit in the highest-numbered bit. The initial coprocessor design required extra circuitry in order to compensate for this conflict. The ESAM bit-ordering would be converted in order to simplify the interface to the remainder of the coprocessor.

3.3.5 Interface Requirements. The design of the interface from the coprocessor to a host processor was based on standard Intel 80386 signals. Actual bus signals and the physical design of the bus interface were unresolved due to a lack of information about the iPSC/2. The following issues would have to be resolved in order to successfully interface the coprocessor:

- Address decoding, which was chosen arbitrarily, would have to be deconflicted with all other I/O devices on the Hypercube.
- Interrupt vector addresses, which were chosen arbitrarily, would have to be deconflicted with other interrupt addresses on the Hypercube.
- The actual hardware interface would have to be determined.

- The bus timing parameters would need to be incorporated into the design of the interface unit.

Since documentation on the Hypercube was limited, these issues would have to be resolved through Intel technical support and customer assistance. Furthermore, although the target architecture was the Hypercube, the interface would have to be designed so that it was flexible enough to support other architectures.

3.3.6 Design Cycle. Figure 4 depicts the design cycle that would be used for this research. Using Daniel's behavioral description of the coprocessor as a starting point, subcomponent descriptions would be modified for the following reasons:

- To convert a behavioral description to a structural description.
- To upgrade a subcomponent (i.e. ESAM)
- To map a description to fit into an FPGA, commercial product, or custom-fabricated circuit.
- To correct an inaccurate description.

After validation of the modified subcomponent, it would be incorporated into the coprocessor description, and the coprocessor would be tested. If the coprocessor worked properly, the cycle would be continued by modifying another component that met the above criteria; otherwise, the same component would be modified until the coprocessor returned correct results.

Validated subcomponents would be implemented as FPGAs, commercial products, or custom-fabricated circuits. Fabricated components and FPGAs would require functional and parametric testing. Finally, the actual timing of the subcomponent would be back-annotated into the coprocessor description.

3.4 Initial Findings

3.4.1 Design Synthesis. Many of the components in the coprocessor could not be fabricated with automated CAD tools and could not be replaced with commercially

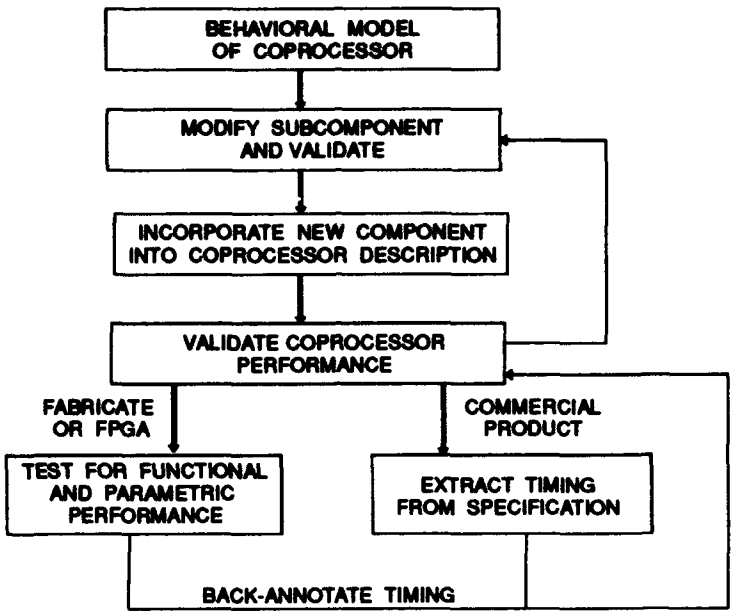


Figure 4. DES Coprocessor Design Cycle

available products without extensive modification to the existing design. For example, the 64 by 32-bit register file could not be synthesized. An attempt was made to synthesize the entire register file; however, this required days of machine time and eventually exceeded the capacity of the available computers. As a test case, a 4 by 32-bit register file with two output port latches required 5172 by 2488 sq μm of area. Therefore, a full-sized register file with additional decode circuitry would easily exceed the maximum size chip area available for fabrication (10,000 by 10,000 sq μm). Furthermore, no three port register files were available in packaging that was suitable for wire wrapping. The use of FPGAs was also not feasible since the most powerful Xilinx FPGA (Xilinx 4020) only had 1,800 flip-flops available in Configurable Logic Blocks and the register file would require 2,048 flip-flops (25:1).

3.4.2 Testbench Results. It initially was not possible to capture all of the coprocessor output because of a lack of documentation on the structure and timing of the output data packets. Once limited coprocessor output was captured, formatted, and written to

file, it became apparent that not all of the results were correct. In some cases, the lowest time stamped message was not being retrieved from the CAM during a Get Event routine. Furthermore, the coprocessor would raise an error condition during a Get Event routine even though it was safe for the LP to process an event. Further analysis revealed that these problems were being caused because data was incorrectly written to the CAM. If data was being written to a reserved word in the CAM, then the write was successful. However, if data was being written to an unreserved word in the CAM, then the message recipient field of the word was being corrupted.

3.4.3 Microcode Routines. Once the microcode preprocessor was written, in-line comments and relative addressing were added to the microcode input file. As a result of this effort, the following errors were detected:

- The Post Event routine was incorrectly identifying the sending LP for null messages.
- The Get Event routine was not functioning properly when a null message was retrieved.
- The microcode did not contain a routine for register file initialization. Further analysis revealed that the register file was being initialized by using VHDL default values.

Finally, the microcode implementation of the SPECTRUM filters did not completely match the current versions being used on the Hypercube. For example, the Time Advance function was incorporated into the Get Event routine; however, time was not updated during the Post Event routine.

3.4.4 Coprocessor Interface Unit. The testbench was rewritten so that it was easier to determine what stimuli were being applied to the coprocessor. As a result, it became apparent that the interface logic between the coprocessor and the host was faulty. In particular, consecutive accesses between the host and coprocessor would cause the two components to deadlock. After each access to the coprocessor, the host would have to access some other device (such as memory) to cause the coprocessor decode circuitry to de-select. The coprocessor would then be ready for another access.

Proprietary documentation on the iPSC/2 Hypercube was obtained from Intel. This documentation indicated that there was available I/O address space for the coprocessor to operate.

3.4.5 ESAM. HSPICE simulations and automated design rule checking revealed the following problems with the Extreme Search Associative Memory:

- Well contacts were incorrectly placed; therefore, V_{SB} would not equal 0 volts.
- The control circuitry incorrectly selected multiple words during single word operations (i.e. read operations and single-word writes)
- Data could not be read from the ESAM. The data output was stuck at 5 volts.
- Equivalence-search operations returned incorrect results.

The first two errors were corrected early in the research; however, the remaining errors were not detected until after the initial design was submitted for fabrication.

3.5 Revised Requirements Analysis

The initial findings caused changes to the requirements and scope of this research. The most significant impact of the initial findings was that none of the previous design work could be treated as a "black-box" since there were so many errors. Implementation of the coprocessor design was not possible since the design was faulty. The scope of the research was changed from implementation to redesign. Partial implementation of the corrected VHDL description would be used to do a performance analysis of the coprocessor.

The following redesign would be required:

- Component descriptions would have to be modified so that they could be mapped into an FPGA, commercial product, or custom-fabricated circuit. Emphasis would be placed on minimizing the chip-count of the coprocessor during this redesign.
- The CAM errors could be ignored since that component was being replaced with the ESAM; however, all of the ESAM errors would need to be corrected.

- The microcode would have to be rewritten in accordance with the architectural changes, and the microcode bugs discussed in the initial findings would need to be corrected.
- The coprocessor interface unit would have to be fixed so that it would allow consecutive accesses from the host. The interface unit would also be modified so that it was not constrained to a particular architecture.

3.6 Conclusion

A thorough analysis of the baseline design was necessary before the initial requirements analysis could be done. Once the initial requirements analysis was done, the first steps were taken towards implementation; however, these steps uncovered numerous problems with the coprocessor design. Once several significant problems were uncovered, a revised requirements analysis was done. The progress of this research was constrained by the problems that were detected with the baseline design. Most, but not all, of these problems would have to be fixed. The scope of the research was changed to redesign the coprocessor so that the VHDL description was correct and feasible. Only partial implementation would be possible; however, a performance analysis could be done based on data collected from the partially implemented design.

IV. Implementation

4.1 Introduction

This chapter explains the completed structural description of the DES Coprocessor using a top-down approach. The intent of this discussion is to give a general understanding of the architecture while explaining design objectives and justifying significant design decisions. Detailed technical data has been intentionally omitted from this chapter and is contained in the appendices.

This chapter also discusses the mapping of the structural description into a physical circuit. Commercial memories, Programmable Logic Devices (PLDs), and custom-fabricated circuits were required in order to map the coprocessor design into a feasible circuit.

Finally, this chapter concludes by discussing the implementation of these subcomponents. Due to time and resource limitations, only portions of the design could be implemented.

4.2 Top-Level Circuit

The top-level design in the structural description of the coprocessor is shown in Figure 5. This level of the design consists of the DES Coprocessor interfaced to a testbench program. The testbench represents a single node on a multiprocessor computer. The testbench provides the stimulus to the coprocessor in order to execute a single-node portion of a parallel discrete event simulation using the Chandy-Misra null-message protocol.

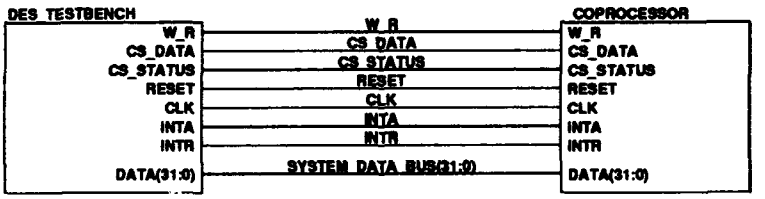


Figure 5. Top-Level VHDL Description

The testbench from the baseline design generated Intel 80386 signals since the target architecture for the coprocessor was the Hypercube. These Intel-specific signals have been eliminated in order to support a more flexible interface. The chip select signals (`CS_Data` and `CS_Status`) are generated directly from the testbench. This configuration allows the coprocessor to be interfaced with varying architectures; however, the logic interface must be completed once a host architecture has been identified. Any additional logic which is required to interface to a particular architecture can be implemented with PLDs.

The testbench provides the coprocessor with opcodes and operands and retrieves results from the coprocessor when appropriate. More detailed information on the testbench-coprocessor interface is in Appendix A. The following opcodes were derived from Reynold's SPECTRUM testbed and are supported by the current design:

- Initialize Simulation - Executed once for each LP that will be simulated on the node. Sets up the internal coprocessor architecture so that the simulation can be successfully synchronized. Processes null messages for each output arc of the initialized LP.
- Post Message - Stores an event for an LP being simulated on the node. Generates a recoverable error condition if the storage capacity of the coprocessor is exceeded.
- Get Event - Retrieves the minimum time-tagged event for an LP being simulated on the node and updates the simulation time of the LP. Generates a recoverable error condition if there is not a valid message on each input arc of the LP.
- Post Event - Processes null messages for the output arcs of the specified LP.

4.3 Internal Coprocessor Architecture

The internal coprocessor architecture is shown in Figure 6. The four major components of the coprocessor are the microcode control engine, an NEQ component, an SRAM, and an interface unit.

The microcode control engine is a 32-bit vertical processor with a programmable control store. The microcode routines are responsible for executing the simulation synchronization routines. The NEQ component is a 32-bit by 32-word Extreme Search Associative

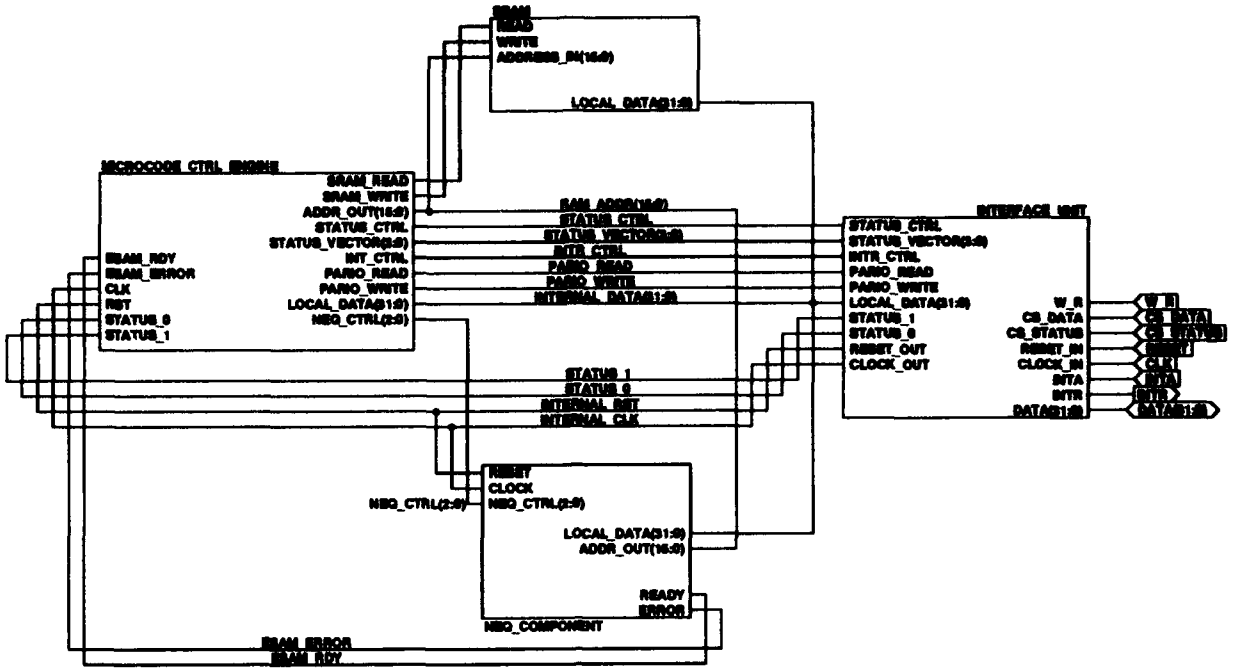


Figure 6. Internal Coprocessor Architecture

Memory with a finite state machine controller and I/O port latches. This component maintains the next event queue for each logical process being supported by the coprocessor. The SRAM is a 64k by 32-bit memory which is used to store the LP specific information required to synchronize the simulation. Finally, the interface unit provides a general purpose I/O interface for the coprocessor. This component allows the host to control the coprocessor and also allows for data transfer between the host and coprocessor.

4.4 NEQ Component

The NEQ Component is shown in Figure 7. This component was included in the DES Coprocessor design to provide O(1) insert and retrieval time of events in the NEQ. As discussed in the requirements analysis, the initial coprocessor design used a CAM within the NEQ component. The CAM was replaced with an ESAM to provide more efficient extreme search capability. Despite the extensive redesign of the component, an effort was made to keep the component interface to the coprocessor the same as the initial interface to minimize the amount of architectural changes that would be required.

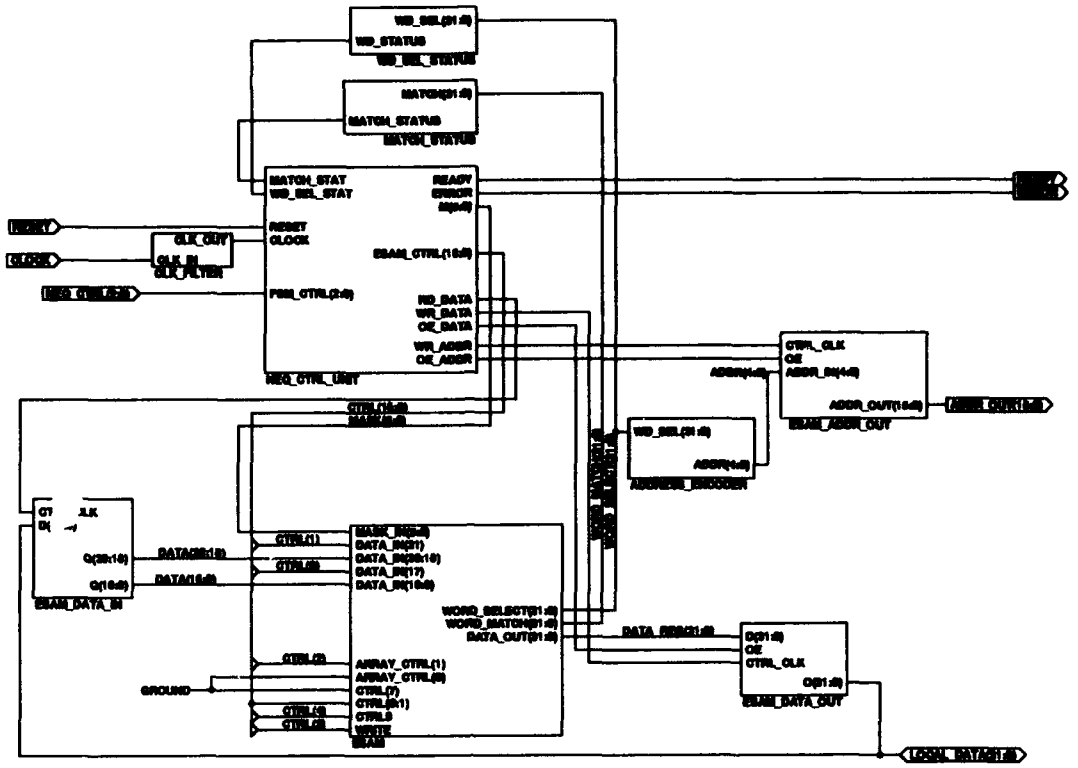


Figure 7. Next Event Queue Component

The NEQ Component consists of the following major subcomponents:

- An Extreme Search Associative Memory (**ESAM**) which can read and write data based on the results of minimum, maximum, and equal searches across the entire memory or subsets of the memory.
- An NEQ control unit (**NEQ_CTRL_UNIT**) which consists of a finite state machine (FSM) with registered outputs.
- An address encoder to convert an ESAM word address to an SRAM address.
- I/O port latches (**ESAM_DATA_IN**, **ESAM_DATA_OUT**, and **ESAM_ADDR_OUT**) to transfer data and address information.
- A clock filter (**CLK_FILTER**) to synchronize the NEQ Component with the multi-phased clock of the control engine.

- Components to provide the FSM with feedback results from the previous operation. The **MATCH_STATUS** and **WD_SEL_STATUS** feedbacks are the "or" of the corresponding 32 signal lines.

The NEQ component does not store all of the event data on-chip. The component maintains the identification of the sender and receiver of the message, the time stamp of the message, and a 32-bit memory pointer to the event. The memory pointer gives the location of the event data which is stored in the main memory of the host processor. This design is based on a distributed memory architecture.

4.4.1 NEQ Control Unit. The NEQ Control Unit controls the I/O port latches, the reading and writing of data in the ESAM, and all search operations. The NEQ Control Unit was designed with the following intentions:

- Off-load the complexity of ESAM control from the DES Coprocessor control engine.
- Allow for concurrency between the control engine and the NEQ Component.
- Allow the capability for the NEQ Component to work as a stand-alone I/O device separate from the DES Coprocessor. This capability would allow the NEQ Component to support both sequential and parallel discrete event simulations.
- Maintain enough flexibility to support both conservative and optimistic parallel simulation paradigms.

The NEQ Control Unit has 42 unique states. A simplified state diagram is in Figure 8. The top portion of the diagram represents start-up states which must be executed to guarantee successful operation. The bottom portion of the diagram represents states which are regularly executed once start-up is complete. Each state in the diagram represents between three to twenty actual states. The control stimulus that causes a transition between top-level states is shown on the arcs. This stimulus is applied to the **NEQ_Ctrl(2:0)** input port shown in Figure 7. More detailed state information is in Appendix B.

A short description of each of the top-level states is listed below:

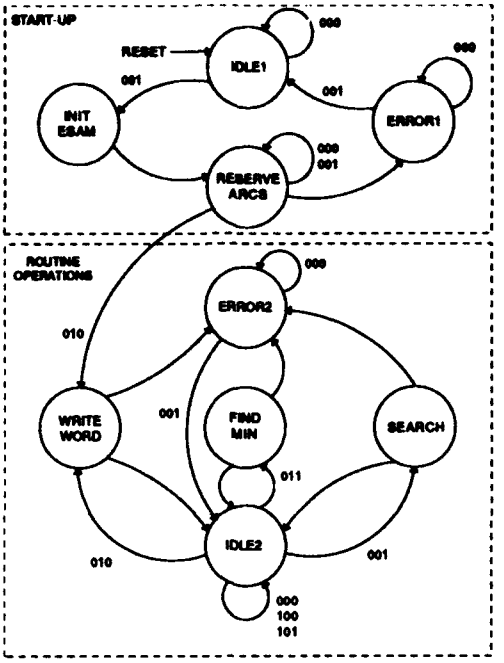


Figure 8. Simplified State Diagram of NEQ Component

- Idle1 - Waiting to begin initialization upon assertion of a Synchronous Reset. This state was necessary because the host (the control engine in this case) may not be immediately ready to begin initialization after a reset.
- Init Esam - Initializes all of the words in the ESAM to a user-defined value which is input through the **ESAM_DATA_IN** port. This state is used to clear all of the valid bits before beginning normal operation.
- Reserve Arcs - Allows words to be reserved for each LP that is being simulated. If multiple LPs are being simulated, this feature guarantees that each LP may have a valid event posted on each input arc. Goes to the Error1 state if an attempt is made to reserve more words than are available in the ESAM.
- Write Word - Writes a word to the ESAM and sets the valid bit. Goes to the Error2 state if the ESAM is full when trying to write. Routine operations begin the first time this state is entered.

- Find Min - Finds, retrieves, and invalidates the minimum time-tagged event for an LP. Goes to the Error2 state if there are no valid events for the LP.
- Search - Searches the ESAM to determine if there is a valid event for the specified LP input arc. Goes to the Error2 state if there is not a valid event on the input arc.
- Idle2 - Waiting to execute another routine operation. May also output data and address information from the I/O ports.
- Error1 - Signals that a non-recoverable error has occurred because an attempt is being made to reserve more words than are available.
- Error2 - Signals that a recoverable error has occurred.

This design will support both optimistic and conservative parallel simulation protocols. For conservative protocols, status registers must be maintained (outside of the NEQ Component) to determine when it is safe to retrieve an event from the NEQ. If it is safe to retrieve a message, the "Find Min" state is entered and the minimum time-tagged event is retrieved. Finally, the "Search" state is entered to determine if there is another valid event on the input arc from which a message was just retrieved. If there is another valid event on this input arc, then the status register does not need to be updated. Otherwise, the status register must be updated to reflect that it is not safe to retrieve another message for this LP.

For optimistic protocols, an attempt to retrieve an event can be made without knowledge of the status of the input arcs; therefore, status registers are not needed. The "Find Min" state is entered to retrieve the minimum time-tagged event for the LP. If there are no valid messages for that LP, then the recoverable error state is entered. The "Search" state is not immediately necessary for optimistic protocols; however, this state may be used for less optimistic approaches which aim to limit aggressiveness or incorporate adaptability.

4.4.2 Extreme Search Associative Memory. The ESAM is shown in Figure 9. This schematic represents the topology of the Magic layout that was provided by Banton from his doctoral research (1). More detailed information on this portion of the design is also in Appendix B. The primary components of the ESAM are:

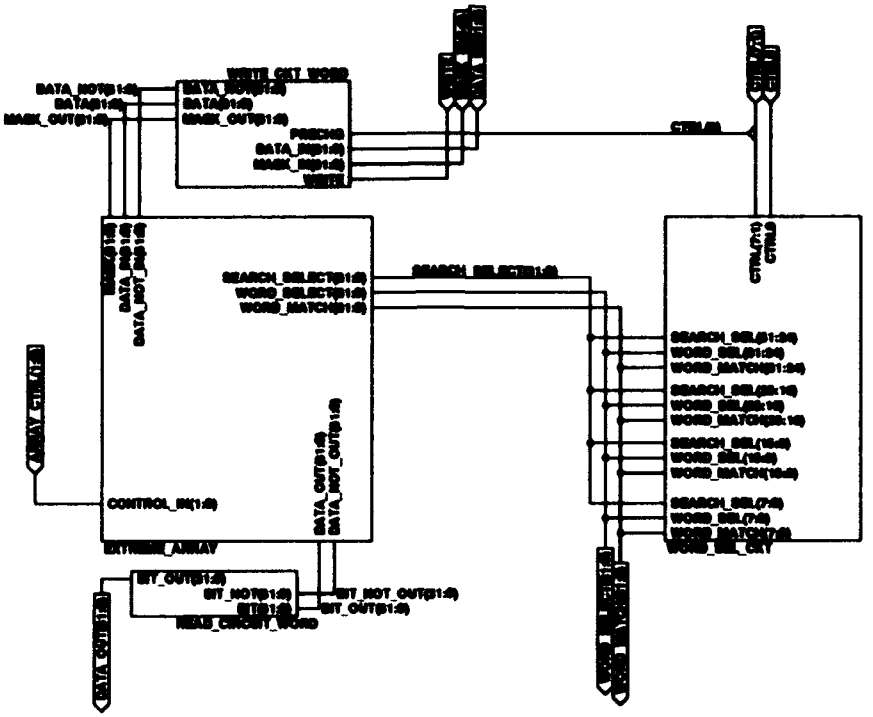


Figure 9. Extreme Search Associative Memory

- A write circuit (**WRITE_CKT_WORD**) which controls the data and mask inputs to the memory.
- A 32-word by 32-bit array of memory cells (**EXTREME_ARRAY**).
- A read circuit (**READ_CIRCUIT_WORD**) which is used to output data that is stored in the array.
- A control circuit (**WORD_SEL_CKT**) which generates the control signals for all write, read, and search operations in the memory array.

The ESAM uses bit-serial word-parallel search logic to execute equivalence, maximum, or minimum searches across all words or subsets of words in the memory. The ESAM can also execute write-all, write-subset, write-word, and read-word operations. Furthermore, all operations can be performed across an entire word or selected parts of a word by using the mask input. Only the search-all-equal and search-subset-minimum associative operations were required for this application, so some of the signal lines in the VHDL

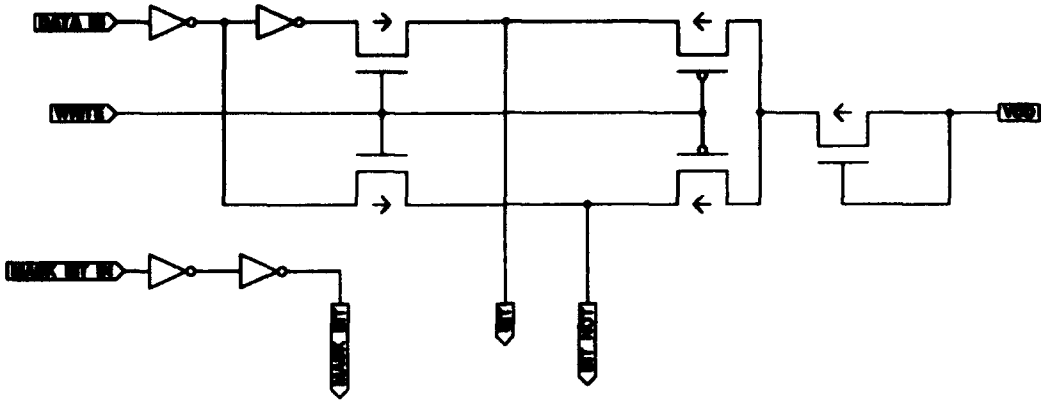


Figure 10. Initial ESAM Write Circuit

description have been tied low. This was not done for the fabricated ESAM components, however, so that these features could still be tested.

Several errors in the initial ESAM design were detected and corrected. These errors are discussed in the following sections.

4.4.2.1 Data Write Circuit. The initial ESAM write circuit for one bit slice is shown in Figure 10. During a write operation, the Write signal is held at 5V, and the precharge portion of the circuit is cut-off. During a read operation, the Write signal is held at 0V, and the precharge is held on for the entire operation. This design allowed data to be written to memory; however, the memory cells could not over-power the precharge drivers sufficiently to allow data to be read from memory.

The write circuit was replaced with the design shown in Figure 11 to correct this problem. This design executes a write operation in the same fashion as the initial design; however, the read operation has been broken into two phases. In the first phase, the precharge signal is held at 5V, and the ESAM bit and bit-not lines charge to approximately 2.5V. Since the bit and bit-not lines are shorted together during this phase, they will precharge to the same voltage. This short reduces the possibility of reading an incorrect value. During the second phase of the read, the precharge line is disabled and the selected memory cells can drive the data lines to the correct voltage levels. To minimize the impact of this change, the precharge signal was derived from an already-existing signal within

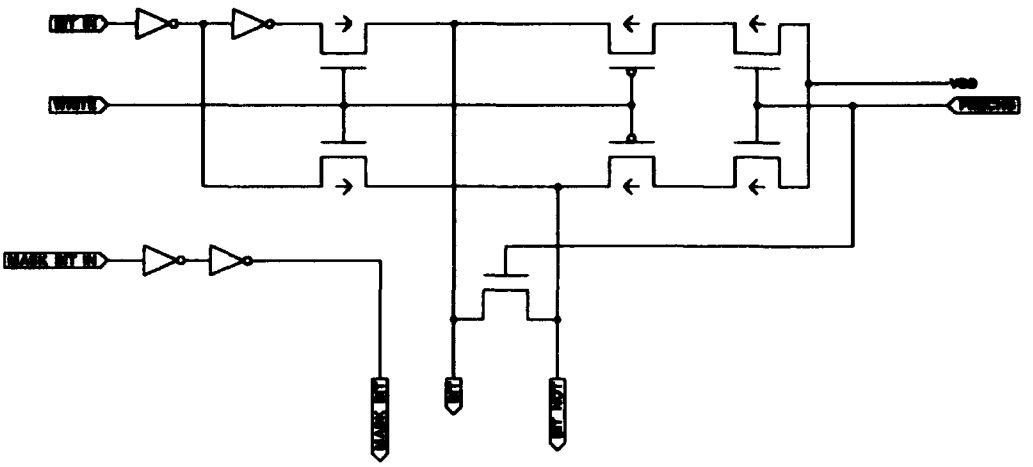


Figure 11. Corrected ESAM Write Circuit

the ESAM design. Therefore, no additional external signals were required because of this modification.

4.4.2.2 Data Read Circuit. The Data Read Circuit is a differential sense amplifier which detects small voltage differences between the bit and bit-not lines and outputs the correct logic value from this information. The initial read circuit of the ESAM was functional; however, it had a small logic swing (less than 1V). To increase the logic swing of the memory, the initial read circuit was replaced with the read circuit used in SanGregory's dual-ported SRAM (23). This modification provided a full five-volt logic swing from the output of the differential sense amplifier and also gave better drive capability for off-memory parasitic capacitances. The new read circuit also has an output-enable control in order to minimize current drain. Since the initial circuit did not have an output-enable, this signal was wired to Vdd to remain asserted. Future research may improve on this; however, a detailed analysis on current drain should first be conducted. Furthermore, the impact on the circuit speed should be analyzed before incorporating the output-enable into the design.

4.4.2.3 Word Select Circuit. The initial word select circuit had several errors which caused multiple words to be selected during single-word operations. The

initial design was correct; however, it was implemented incorrectly. Several subcomponents of the control circuitry were either wired together incorrectly or had dangling wires. These errors were detected and corrected.

4.4.3 Adjacent SRAM. Not all of the NEQ data is used for associative search operations. For example, the memory pointer to the event data is not searched. Since the ESAM memory cell requires 26 transistors, it would be inefficient to use this type of memory cell for the sole purpose of storing data. It would be much more efficient to use an SRAM cell composed of only six transistors to store data that is not part of the search.

Ideally, this Adjacent SRAM would be included on the same integrated circuit (IC) as the ESAM. For ease of implementation, an address encoder and address port register were included in the NEQ Component so that a separate commercial SRAM could be used to store this adjacent data. Refer to Figure 7. The 32 ESAM words are encoded into a 5-bit address field; however, the NEQ Component outputs a 16-bit address in order to conform to the SRAM memory map. This simplified design will result in a slower over-all execution time to account for the address encoding and driving of chip-to-chip capacitance.

4.5 Microcode Control Engine

The microcode control engine is shown in Figure 12. The control engine is composed of two major components: an execution unit and a control unit. The execution unit manipulates data. The control unit controls the execution unit, NEQ Component, SRAM, and Coprocessor Interface Unit. Detailed information on the Microcode Control Engine is in Appendix C and Appendix D.

4.5.1 Control Engine Execution Unit. The execution unit is shown in Figure 13. The Execution Unit consists of the following major components:

- A 16 by 32-bit general-purpose register file (`GPR_File`) with two register address decoders (`R1_DECODER` and `R2_DECODER`) to select registers for read/write operations.
- A 32-bit ALU and shifter for manipulating data.
- A 32-bit bidirectional memory buffer register (`MBR`).

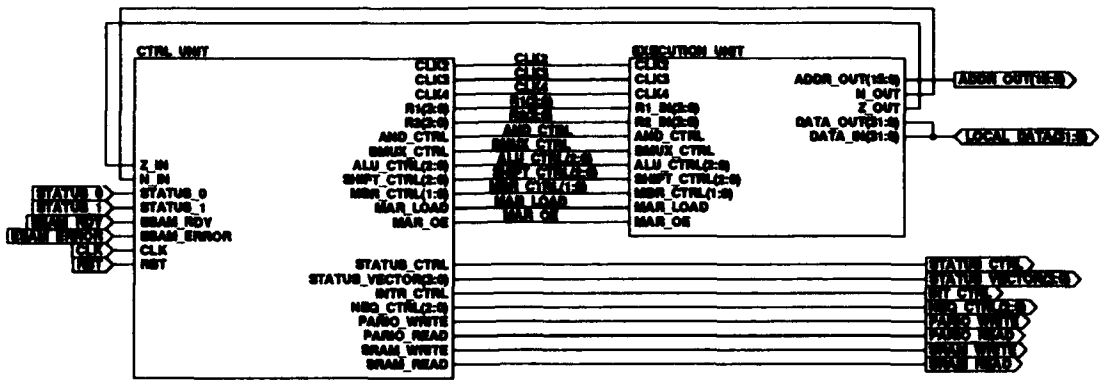


Figure 12. Microcode Control Engine

- A 16-bit output memory address register (**MAR**).
- Two data path latches (**A_LATCH** and **B_LATCH**) and a multiplexer (**BMUX**) to control the internal data path of the execution unit.

4.5.1.1 General Purpose Register File. The initial coprocessor design required a 64 by 32-bit general purpose register file. A register file of this size could not be purchased or fabricated; therefore, architectural and microcode changes were necessary in order to reduce the number of required registers. This reduction was accomplished by trading off speed for size.

An encoding scheme was used to eliminate the need for twenty of the general purpose registers. In the initial design, an SRAM pointer for each LP was kept in the register file. Since the coprocessor can accommodate up to twenty logical processes, twenty dedicated registers were required to maintain pointers to the respective LP partition in the SRAM. The SRAM pointer for each LP is simply the five-bit LP identifier shifted left five times to allow an LP partition of thirty-two words. Since the LP identifier is provided in every opcode to the coprocessor, the LP SRAM pointers could be decoded from the opcodes thus eliminating the need for dedicated registers.

The size of the register file was further reduced by moving data to the SRAM. The initial design used registers to maintain the status of the input arcs for each LP. Twenty dedicated status registers were required for this initial scheme. The LP status information

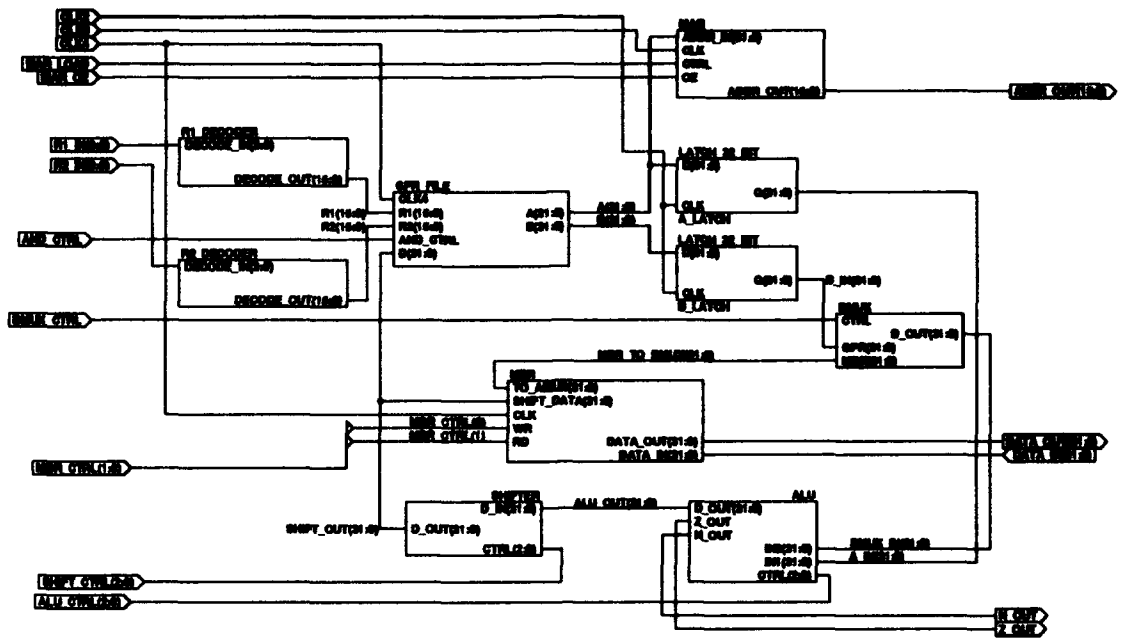


Figure 13. Execution Unit of Microcode Control Engine

was moved to storage in the respective LP partition in the SRAM. When the LP status needs to be updated, it is transferred from SRAM to register, updated, and then written back to SRAM. This modification eliminated the need for twenty registers.

Of the remaining twenty-four registers, less than sixteen were required to execute the current microcode routines. The size of the register file was chosen to be sixteen to conform with a power-of-two multiple.

The primary trade-off with the redesign of the register file was between speed and size. Since the LP SRAM address would need to be decoded and status information maintained in SRAM, execution time would be slower, but fewer registers would be required. The impact of this trade-off could be minimized by modifying the format of the user-defined macroinstructions; however, the initial format was not modified due to time limitations. The impact on the microcode performance is listed below.

- Decoding of the SRAM address added 24 clock cycles to each microcode routine.
- Maintenance of the LP status in the SRAM added 20 clock cycles to the Post Message routine.

- Maintenance of the LP status in the SRAM added 32 clock cycles to the update function within the Get Event routine.

The average time increase of the microcode routines is dependent on simulation specific variables which are not discussed until Chapter V. Typical microcode routines should require several hundred clock cycles to execute. The over-all impact of the smaller register file should be a time increase of less than 10 percent.

The utility of the architecture was not sacrificed by reducing the size of the register file. In comparison to other microcode control processors, both the Intel 80486 and Motorola 68030 have only eight general purpose data registers (12, 18). Sixteen general purpose registers combined with the coprocessor SRAM should be sufficient for the DES Coprocessor to execute potential synchronization tasks.

4.5.1.2 ALU. The functionality of the ALU was improved from the initial design to compensate for a smaller register file and to increase the utility of the coprocessor. The following functions were added:

- Increment and Decrement - These functions eliminated the need to use registers to store values of $\pm$ 1.
- Pass-through of Source Register - This function allowed for register-to-register moves. The initial ALU could only pass the destination register, so a register-to-register move required two cycles:

  1. "and" the target register with zero to clear it.
  2. "or" the target register with the source register.

  This ALU modification cut the execution time of the most common instruction in half and eliminated the need for a zero-register.
- Subtract - Although not immediately needed, the ability to subtract two numbers was added to the ALU to improve the utility of the coprocessor. This was a low-cost improvement to the ALU since the the subtracter was already in place to support the Decrement function.

The current implementation of the adder/subtracter portion of the ALU is a carry-ripple. This implementation was selected since it could be easily synthesized in the Synopsys Design Analyzer.

4.5.2 Control Engine Control Unit. The Control Unit is shown in Figure 14 and consists of the following major components:

- An EPROM control store which outputs 16-bit microinstructions.
- A microinstruction decoder with registered output (**MICROINSTRUCTION_DECODE_UNIT**).
- A mapping multiplexer (**MMUX**), microsequencing logic (**MSL**), and a carry-ripple adder (**INCREMENTER**) for generating the address of the next microinstruction.
- A branch target latch (**BRANCH_TGT_LATCH**) which stores the absolute address of a branch target specified by the microinstruction. If a branch is not specified, this latch stores the encoded R1/R2 fields from the microinstruction.
- A microprogram counter (**MPC**) that holds the address of the next microinstruction. "CLK4" is used to load the MPC, and "CLK1" is used for a synchronous reset.
- A four-phased non-overlapping clock generator (**CLOCK**) to control the engine data path and prevent race conditions.
- A flag register (**FLAG_REG**) to store the results of the ALU operation from the previous instruction.

4.5.2.1 Programmable Control Store. The initial design used an SRAM for the programmable control store and required a bootstrap ROM to control the loading of microcode. The control store was moved to an EPROM in order to decrease the complexity of the design. The major benefits of using an EPROM control store are:

- Simplified data path and control circuitry since the need for a run-time load of the control store was eliminated.
- Reduced frequency of control store programming since reprogramming would be required only when there was a change in the microcode.

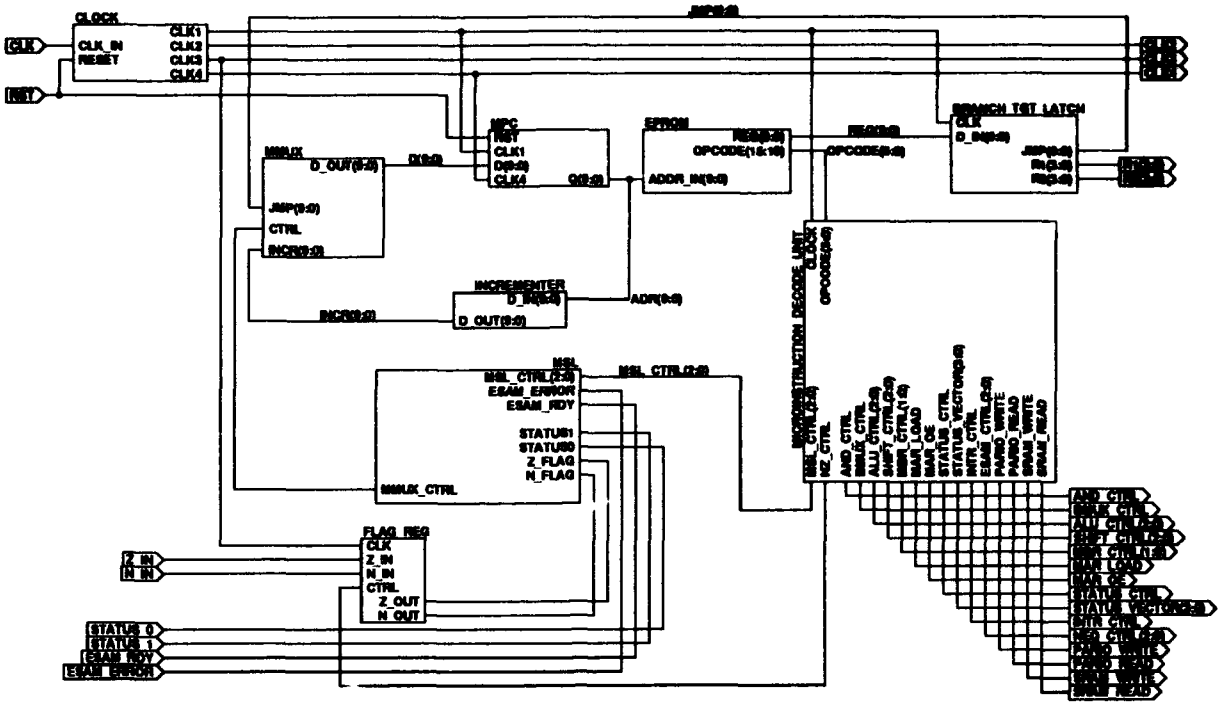


Figure 14. Control Unit of Microcode Control Engine

- Faster VHDL simulation time since the control store load did not need to be simulated before testing the design. The simulation time was reduced by over 5 minutes. This reduction resulted in faster design iterations.

The only major cost of this redesign is that the control store could not easily be placed on the same IC with other components of the coprocessor. Therefore, the over-all IC count of the coprocessor would be higher.

4.5.2.2 Macroinstruction Decoding. The initial design also had a Mapping SRAM so that the instruction register in the general purpose register file could be used to select a branch target in the microprogram. This allowed for a fast decode time during the fetch/decode cycle in the microcode; however, it required complex data path and control circuitry since the Mapping SRAM had to be loaded at start-up. Furthermore, the Instruction Register had to be hard-wired to the mapping-multiplexer which would limit the flexibility of the register file and also further complicate the data path. To reduce this complexity, the design was modified to decode instructions by shifting the instruction

and passing it through the ALU. This modification added an average of 16 clock cycles to the fetch/decode routine and increased the average microcode execution time by less than 5 percent.

4.6 SRAM Component

The initial coprocessor design had a 4k by 32-bit SRAM which was used for holding LP specific information needed to synchronize the simulation. This initial design had a predefined memory map in which the microinstruction decoder was used to select one of four 1k by 32-bit memory partitions. The address within the 1k partition was then selected by the 10-bit Memory Address Register (MAR).

To improve the utility of the coprocessor, this addressing scheme was changed so that the memory map could be completely defined by the user. Rather than specifying part of the address from the microinstruction decoder, the entire address is now specified from the MAR.

The size of the SRAM was also increased to improve the utility of the coprocessor. The permissible SRAM size was increased from 4k to 64k. This was accomplished by increasing the MAR size to 16 bits. This increase in memory size was not based on a particular need, but was aimed at increasing the flexibility of the coprocessor.

Another change from the initial design was to share the SRAM between the Control Engine and the NEQ Component. The Control Engine uses the SRAM to maintain partitions of LP specific information which is required to synchronize the simulation. The NEQ Component uses the SRAM to store the memory pointer to event data. In the current design, the NEQ Component and Control Engine do not simultaneously access the SRAM, so a single-port memory is sufficient. The sharing of the SRAM between the Control Engine and the NEQ Component was done to minimize the IC count of the coprocessor. The memory map for the current implementation is given in Appendix E.

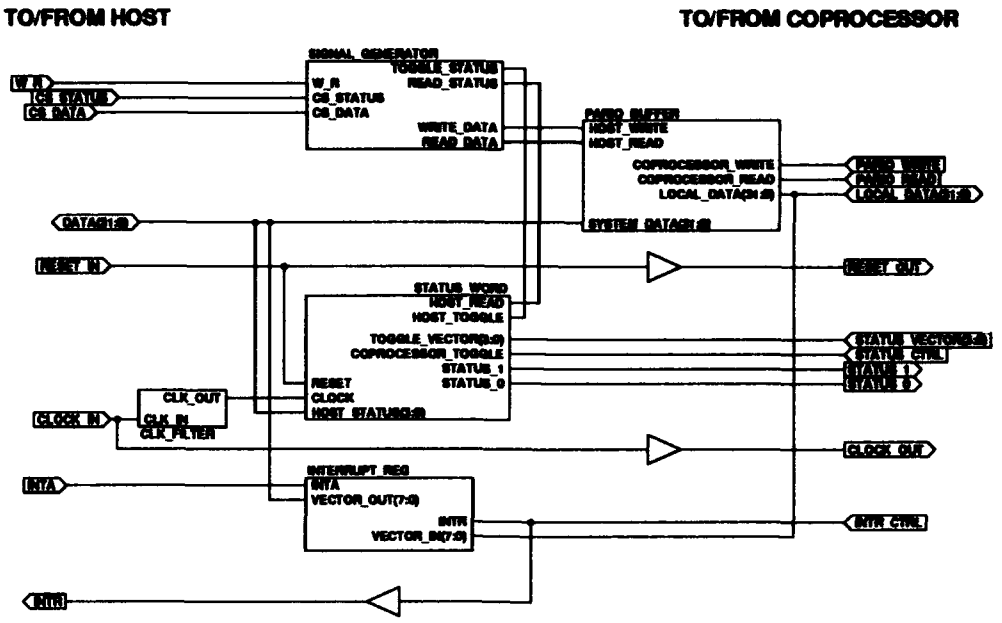


Figure 15. DES Coprocessor Interface Unit

4.7 Interface Unit

The interface unit is shown in Figure 15. Detailed information on this component is in Appendix F. The Interface Unit consists of the following major components:

- A Signal Generator which is used to interpret the control stimulus from the host processor and generate the correct internal signals.
- A 32-bit parallel I/O buffer (`PARIO_BUFFER`) which is used to transfer data between the host and the DES coprocessor.
- A four-bit status register (`STATUS_WORD`) which is used by both the host and coprocessor to transfer status information.
- An eight-bit interrupt register (`INTERRUPT_REG`) which is used by the coprocessor to provide interrupt vectors to the host.
- A clock filter (`CLK_FILTER`) to synchronize the status register with the multi-phased control clock of the control engine.

Table 1. Mapping of Coprocessor into Physical Circuits

Component	Method of Implementation
NEQ Component	Custom-fabricated
Ctrl Engine Execution Unit	Custom-fabricated
Ctrl Engine Ctrl Store	Commercial EPROM
Ctrl Engine Ctrl Unit (less Ctrl Store)	FPGA
SRAM	Commercial SRAM
Interface Unit	FPGA

The Interface Unit in the initial coprocessor design was based on Intel 80386 signals. The design was modified so that the coprocessor could be used as a general-purpose I/O device for any 32-bit host processor. Two chip select signals (`CS_Data` and `CS_Status`) in conjunction with a Write/Read (`W_R`) signal allow the host to modify or read data in the Parallel I/O Buffer and Status Register, respectively.

4.8 Mapping of Design into Physical Circuits

Table 1 shows the mapping of the structural description of the coprocessor into actual components. The following criteria were used to complete this mapping:

- Use FPGAs to the maximum extent possible to maintain flexibility in the design while reducing risk and cost.
- Use other commercial products if they are readily available and can be used without extensive modification to the design.
- Use custom-fabricated circuits only as a last-resort because of high risk and cost.

The only aberration from the mapping criteria is the Control Engine Execution Unit. Although this component may possibly be implemented on an FPGA, the required equipment was not available. Furthermore, this portion of the coprocessor is the least likely to change. Because of these considerations, this portion of the design was implemented on a fully-synthesized custom-fabricated circuit. Once the required equipment does arrive, an attempt can be made to fit this portion of the design on a single FPGA. If the design fits

on a single FPGA, the custom-fabricated design and the FPGA design can be compared to choose the fastest implementation.

4.9 Implementation of Subcomponents

Due to time and resource constraints, the design was only partially implemented. The following subsections discuss the different categories of implementation.

4.9.1 Field Programmable Gate Arrays. The Xilinx FPGA family was chosen because of its compatibility with the Synopsys CAD tools. The Synopsys Design Analyzer could be used to convert either behavioral or structural VHDL descriptions to the initial format that is used by the Xilinx software. Unfortunately, Synopsys only supports the Xilinx 4000 family, and the PC-based Xilinx software that was available only supports the Xilinx 2000/3000 families. The PC-based software could not be used because there was no automated process to translate the VHDL descriptions to the correct format for the Xilinx software. Manual translation was not feasible because of time constraints. The Sparc-based Xilinx software which supports the 4000 family was ordered; however, this software was not available in time to support this research.

4.9.2 Commercial Memories. Commercial memories were chosen for the SRAM and the Control Engine Control Store. The following criteria were used to select the correct memory:

- The memory packaging must be suitable for wire-wrapping. Many of the high-performance memories on the market come in packaging which is only suitable for printed circuit board designs. Pin spacing of the package was the primary factor.
- The memory speed should be as fast as possible, and the memories should be readily available.
- The memory size (on one package) should meet the VHDL description in order to minimize the coprocessor IC count.
- The EPROM must be programmable with equipment available in the AFIT Advanced Processor Lab.

The Cypress CY7C261 was selected for the EPROM. Although a 16-bit EPROM is required to place the control store on a single IC, the available equipment is not capable of programming such an EPROM. An 8-bit EPROM was chosen, so two packages will be required to implement the control store.

The Cypress CY7C198 was selected for the SRAM. Four packages will be required since this SRAM is only 8 bits wide. Furthermore, the Cypress SRAM has only 32k of words, so it is not possible to use the full address range of the coprocessor without additional logic. The coprocessor can accommodate up to 64k; however, 32k is sufficient for the current implementation.

4.9.3 Fabricated Components. The NEQ Component and Control Engine Execution Unit were the only two components of the coprocessor selected for custom fabrication. Since the NEQ Component could not be simulated with a switch-level simulator, it was necessary to fabricate subcomponents before attempting to fabricate the entire component on one IC. Fabricating the subcomponents provided sufficient controllability and observability to identify design errors. Once all of the fabricated subcomponents were validated, they could be incorporated into a single IC. Pin assignments of all fabricated circuits are listed in Appendix G. The following parts of the NEQ Component were fabricated:

- Memory Array - A 32-word by 32-bit ESAM array with write and read circuitry was fabricated in a 132-pin Pin Grid Array (PGA). The control portion of the ESAM was omitted from this circuit, so all control signals to the array were brought in from off-chip. Furthermore, the result signals from search operations were sent off-chip. As discussed in Section 3.4.5, several errors were detected with this IC after it was submitted for fabrication. Despite these errors, the extreme-search functions could still be tested.
- Word Select Circuit - A 32-word ESAM word select circuit was fabricated in a 132-pin PGA. The ESAM memory array was not included in this package, so the memory array control signals were sent off-chip and the search result signals were brought in from off-chip.

- NEQ Control Unit - The NEQ Control Unit was fabricated as a MOSIS TinyChip in a 40-pin Dual In-Line Package (DIP).
- Corrected ESAM Array - A 4-word by 6-bit ESAM memory array with corrected read/write circuitry was fabricated as a MOSIS TinyChip in a 40-pin DIP.

The execution unit of the control engine was fabricated in an 84-pin PGA. The Magic layout was generated from the VHDL structural description by using the Synopsys Design Analyzer and the Octools/Lager software package. Furthermore, limited switch-level simulation of this circuit was possible prior to fabrication. Because of these factors, subcomponent fabrication was not necessary prior to fabricating the entire execution unit.

4.10 Conclusion

This chapter provided an overview of the completed structural description of the DES Coprocessor using a top-down approach. The intent of this discussion was to give a general understanding of the architecture while explaining design objectives and justifying significant design decisions. This chapter also provided an explanation of the criteria used for mapping the structural description of the coprocessor into physical circuits.

The final portion of the chapter explained the partial fabrication of the coprocessor. Since the NEQ Component could not be simulated with a switch-level simulator, subcomponents were fabricated on separate ICs to provide sufficient controllability and observability for design validation. These fabricated ICs would also provide enough timing data for a partial performance analysis of the coprocessor.

V. Findings

5.1 Introduction

This chapter reviews the test results of the custom-fabricated components of the DES Coprocessor. More detailed information on the testing of the circuits is in Appendix H. Timing measurements from these tests were back-annotated into the VHDL description of the coprocessor. The critical path and maximum operating frequency of the coprocessor were determined from this analysis.

Equations are derived to determine the coprocessor performance based on application specific variables. Finally, a partial performance analysis of the coprocessor is done using data extracted from Hypercube simulations with the AFIT Algorithm Animation Research Facility (AAARF).

5.2 ESAM Array

5.2.1 Functionality Test. The memory array I/O functions and associative operations were tested in order to validate the design. With one exception, the results of this testing were consistent with the predicted functionality from HSPICE simulations. As expected, data could be written to the array and extreme search operations could be executed. Equivalence operations returned incorrect results because an inverter on the comparand data path was not connected to the power line. In some cases, data could be read from the array, and this observation was a departure from the HSPICE predictions. Based on the discussion in Section 3.4.5, the failure of the equivalence search operations and the read operations can be attributed to design errors and not fabrication errors. Thus, the fabrication yield was 100 percent for the 32 ICs that were fabricated.

5.2.2 Extreme Search Test. Worst-case maximum and minimum searches were executed across all 32 words using search fields of 32, 31, 28, 24, 20, 16, 12, 8, and 4 bits (Figure 16). The size of the search field was constrained by the number of pins on the package. In order to place the array in a 132-pin PGA, one mask input on the IC corresponds to a 4-bit field. Although the maximum search is not required for the DES

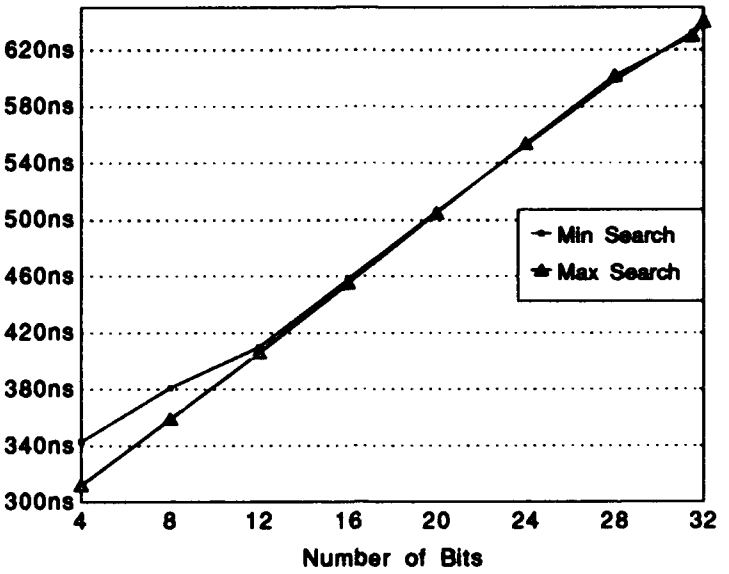


Figure 16. Worst-Case ESAM Operations using a 32-bit by 32-word memory. The searches were executed across all 32 words, and the width of the search field was varied. Time is measured from the assertion of search select until the return of word match as shown in Figure 9.

Coprocessor design, this function was measured in order to provide a comparison with the minimum search function. These functions should have near-identical performance, and any differences might identify flaws in the design.

All 32 ICs were tested for functionality; however, only 10 ICs were tested for worst-case performance due to the complexity and time requirements of the testing. This testing is discussed in greater detail in Appendix H. The worst-case minimum (maximum) test was executed as follows:

1. Write FFFFFFFF (00000000) to all words in the memory.
2. Write 00000000 (FFFFFFFF) to a single word in the memory.
3. Search all words in the array for the minimum (maximum) value.

This test was the worst-case scenario for extreme searches since only one memory cell must drive each slice of the extreme search data path. The searches were executed

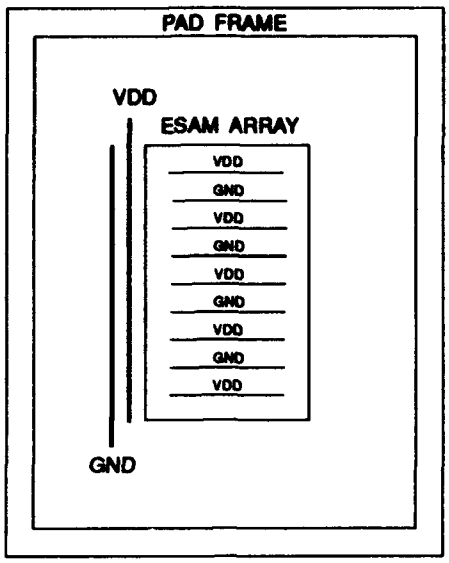


Figure 17. Simplified ESAM Array Floor Plan

across all 32 words; however, the results are the same for subset searches. The capacitive loading of the search path is not affected by the number of words selected.

Figure 16 reflects the linear relationship between the width of the search field and the time required to execute an extreme search. The minimum and maximum searches are executed in equivalent time (±3ns) except when the width of the search field is less than 12 bits. Further analysis revealed that this difference may be caused by the poor power and ground routing of the circuit. Figure 17 shows a simplified floor plan of the fabricated circuit. Primary power and ground rails are used on the side of the circuit which represents the high-order bits. Memory cells in the low-order bits must rely on the power and ground rails on the opposite end of the layout.

The poor power and ground distribution probably has a more significant impact on the minimum search data path and causes the difference in execution times between minimum and maximum searches. When the worst-case extreme search was executed on a 4-bit field using the high-order bits of the ESAM words, the search times decreased,

Table 2. Worst-Case ESAM Operations across a 4-bit search field. When the high-order bits are searched, the maximum and minimum search times are reduced and become comparable. Units are in nanoseconds; σ is the standard deviation. Times are mean values taken from the same 10 ICs used in Figure 16.

Search	Bits 27-24	σ	Bits 3-0	σ
Maximum	291.7	5.3	312.1	6.6
Minimum	297.6	6.0	343.5	6.2

and the difference in execution time between the minimum and maximum searches was reduced (Table 2).

The standard deviation of the worst-case maximum and minimum searches is plotted in Figure 18. As the width of the search field is increased, the standard deviation also increases; however, the standard deviation divided by the mean remains constant at 2 percent.

5.2.3 Write-Read Test. The I/O performance of the ESAM array did not completely match the predictions from HSPICE simulations. In general, the data output of the memory array was stuck at a high logic value; however, in a few cases, the correct data could be read from the array.

If the same data was written to two words and both words were simultaneously read, then the correct data was output for 69 percent of the ICs. This result indicates that the simultaneous selection of two memory cells on the same data line will lower the resistance path to ground sufficiently to pull the precharged line down.

Read operations for single words were successful for 19 percent of the ICs. Timing data for these ICs was not collected because the results were inconsistent. For example, the read access time for the same IC could fluctuate by 200ns while running the same test.

Differences between the HSPICE simulations and the actual circuit performance can be attributed to two factors. First, the accuracy of the HSPICE simulations was

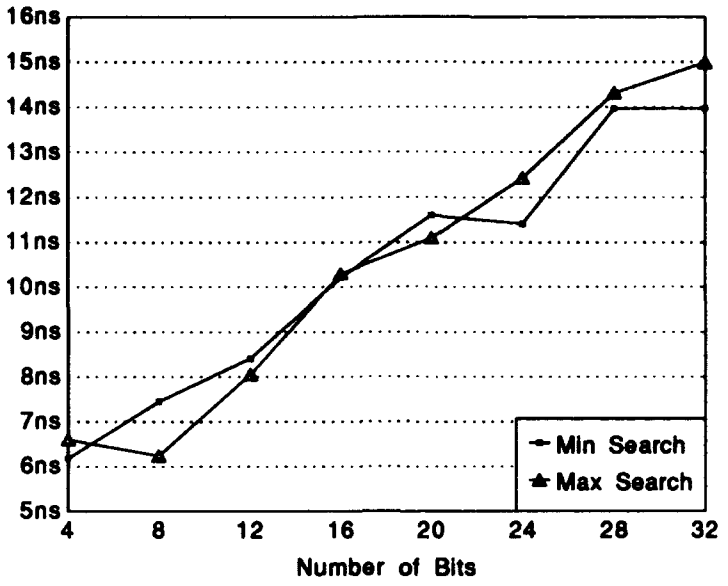


Figure 18. Standard Deviation of Worst-Case ESAM Search Operations shown in Figure 16. σ increases with the number of bits; however, σ/μ is constant at 2 percent.

intentionally sacrificed in order to decrease the execution time of the simulation. The HSPICE `fast` option was used to reduce the simulation time to 8 hours at the expense of accuracy. Furthermore, the noise margin of the differential sense amplifier was small enough that an accurate prediction of performance was not possible. For 81 percent of the circuits, HSPICE simulations predicted the actual behavior (data output stuck at a high logic value).

5.3 ESAM Word Select Circuit

5.3.1 Functionality Test. Not all of the Word Select circuit functions are required in the DES Coprocessor design. For example, the search-not-min/max/equal and the write subset functions are not used. These functions and the required functions were tested in order to validate the circuit design.

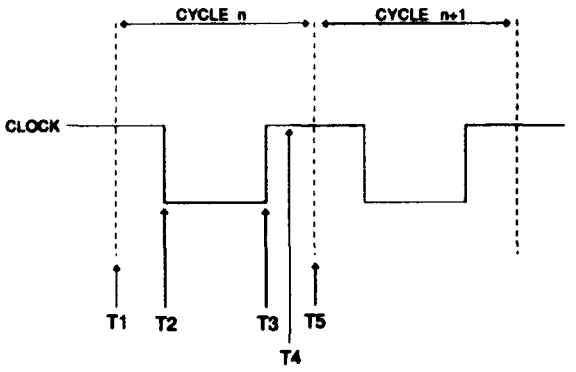


Figure 19. NEQ Control Unit Cycle. The pulse represents the clock input to the NEQ Control Unit. Events T1-T5 are explained in Table 3.

This testing revealed a subtle difference between the VHDL model and the fabricated circuit. Refer to Figure 9. The correct CTRL(3) input for the fabricated circuit is the inverse of the signal for the VHDL model. This difference occurred because an inverting buffer was used to drive the parasitic capacitance on this signal line in the fabricated circuit. After the CTRL(3) input to this circuit was inverted, all of the functions worked correctly. The fabrication yield was 100 percent.

5.3.2 Performance Test. The most time consuming functions of the Word Select circuit are the Search-All and Write-All functions. The mean minimum cycle time of the control inputs was 25.9 ns (38.6 MHz) with a standard deviation of 0.49 ns. The sample size was 32 ICs.

5.4 NEQ Control Unit

The functional and parametric tests for the NEQ Control unit were executed simultaneously. The test consisted of traversing all of the arcs in the finite state machine. Figure 19 and Table 3 explain the performance of this circuit. The NEQ Control Unit operates successfully with a maximum clock rate of 27.6 MHz and a duty cycle of 70 percent.

Table 3. NEQ Control Unit Cycle. Time is relative to the start of a cycle.

Event	Description	Time
T1	Start of cycle. Control stimulus applied to NEQ Control Unit inputs.	0ns
T2	Falling edge of clock input. Registered outputs are latched.	10.5ns
T3	NEQ Control Unit transitions to the next state.	21.3ns
T4	NEQ Control outputs become valid.	24.2ns
T5	Start of next cycle.	36.2ns

5.5 Corrected ESAM Array

The corrected ESAM Array is a 4-word by 6-bit memory. This circuit was fabricated to validate the improved write/read components of the memory and the corrected Equal-Search function.

The worst-case performance of the associative operations for this IC is shown in Figure 20. The worst-case extreme searches were executed as described in Appendix H. The worst-case Equal Search was executed by generating a simultaneous match on all words in the memory.

As can be seen in Figure 20, the Equal-Search operation always performed faster than the extreme searches. Furthermore, as the width of the search field was increased, the difference in execution time between the extreme and equal searches increased. This result is expected since the Equal-Search path has less parasitic capacitance per gate than the Extreme-Search path.

The write and read test of the corrected array consisted of writing and reading patterns of zeroes and ones to each word in the array. Table 4 shows the timing measurements for this test. The cycle time refers to the rate at which the data, mask, write, precharge, and word select inputs were applied to the memory, and the read access time refers to the

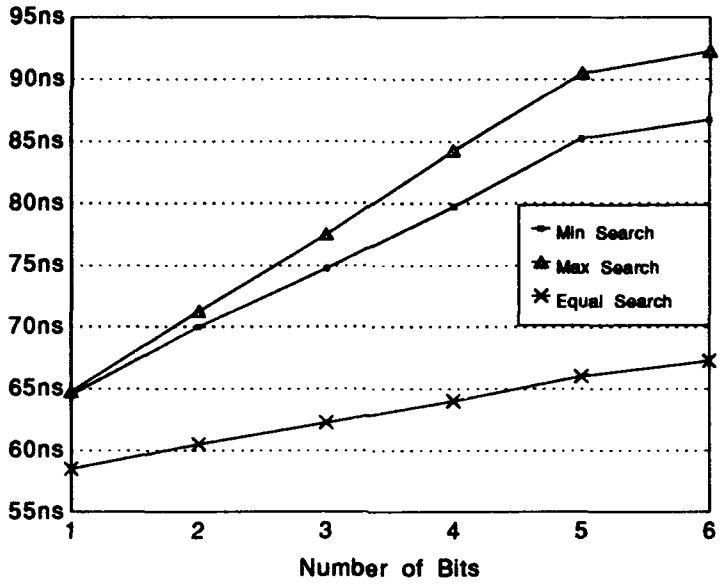


Figure 20. Worst-Case ESAM Search Operations for Corrected ESAM Array. Data points are mean values taken from a sample size of 4 ICs. The standard deviation ranged from 3ns up to 5ns as the width of the search was increased.

Table 4. Write-Read Performance of Corrected ESAM Array

Cycle Time	Read Access
50ns	10.75ns
15ns	8.50ns

time measured from when the word select line was asserted until the output data became valid.

As the cycle time of the control and data inputs was reduced, the read access time became faster. A prolonged precharge phase during a read operation slowed down the read access time because the data lines become over-charged. As the cycle time was reduced, the

precharge duration was reduced and approached the optimal value for a 4-word memory. For larger memories, both the optimal precharge time and the read access time would increase in proportion to the parasitic capacitance of the data lines.

5.6 Coprocessor Critical Timing Analysis

The potential critical path of the DES Coprocessor is based on the timing data from the fabricated components and the timing specifications of the commercial memory devices. The Control Engine Execution Unit and FPGA components were not considered in this analysis.

All timing measurements of the fabricated components include the pad delays. Since several of the tested components may be included on a single IC, the actual timing could be faster since the off-chip parasitic capacitance would be eliminated. Including the pad delays into the timing measurements allows for a conservative estimate of true performance.

Based on this partial analysis, the critical path of the DES Coprocessor is the minimum search path across the 17-bit time field of words in the ESAM. From Figure 16, this operation will require 460ns for a worst-case search. A clock filter is used to synchronize the NEQ Component with the remainder of the coprocessor (Figure 7). Thus, four clock pulses into the coprocessor correspond to one NEQ Component cycle. The minimum clock cycle of the coprocessor is calculated in Equation 1.

$$\left(\frac{460ns}{NEQ\ Component\ Cycle}\right)\left(\frac{1\ NEQ\ Component\ Cycle}{4\ Coprocessor\ Cycles}\right) = \frac{115ns}{Coprocessor\ Cycle} \quad (1)$$

Despite the reduction of the coprocessor clock cycle through the use of a clock filter in the NEQ Component, the critical path remains the extreme search path in the ESAM. Thus, the maximum clock rate of the coprocessor is 8.7 MHz.

Table 5. Application Specific Variables for Coprocessor Performance

Variable	Description
Fetch	Fetch-Decode time of instruction (Table 6).
$Arcs_{In}$	Number of input arcs.
$Arcs_{Out}$	Number of output arcs.
$\%Write_R$	Percentage of writes to reserved words in the ESAM.
$\%Write_U$	Percentage of writes to unreserved words in the ESAM. (1 - $\%Write_R$)
%Null	Percentage of messages that are null messages.
%Real	Percentage of messages that are real messages. (1 - %Null)
%Update	Percentage of times that LP status must be updated after Get Event.

5.7 Coprocessor Performance

The average number of clock cycles required for the coprocessor to execute the synchronization routines for an LP is given by Equations 2 through 5. These equations give the average number of clock cycles measured from when the coprocessor receives an opcode until the result data is returned to the host (if required) or the coprocessor signals that it is ready for another opcode. The equation variables are defined in Table 5.

$$Init\ Sim = Fetch + 200 + 92(Arcs_{In}) + 128(Arcs_{Out}) \quad (2)$$

$$Post\ Msg = Fetch + 196(\%Write_R) + 208(\%Write_U) + 40(Arcs_{In}/2) - 4(\%Null) \quad (3)$$

$$Get\ Event = Fetch + 376(\%Real) + (320 + 96(Arcs_{Out}))\%Null + 40(Arcs_{In}/2)(\%Update) \quad (4)$$

$$Post\ Event = Fetch + 44 + 92(Arcs_{Out}) \quad (5)$$

Table 6. Cycles Required for Fetch-Decode

Routine	Cycles Required
Initialize Simulation	104
Post Message	80
Get Event	92
Post Event	104

Two simplifying assumptions were made to arrive at these performance equations.

Assumption #1 The message traffic on the input arcs of the LP is uniform.
Assumption #2 The host is always ready for result data from the coprocessor.

The first assumption impacts on the performance of the Post Message and Get Event routines. The microcode routine to update the status of an input arc for an LP is executed in O(n) time complexity. Assuming uniform message traffic on all of the input arcs, the average performance of this part of the routine is a factor of n/2.

The second assumption impacts on operations which return data to the host (Initialize Simulation, Get Event, and Post Event). If the host is not ready to receive the data, the coprocessor must remain idle until the host is ready. The average idle time while waiting to return results to the host may be added against the execution time of the routines. This time has been ignored because it would be unfair to penalize the coprocessor execution time because the host is doing other useful work.

5.8 Comparative Performance Analysis

5.8.1 Pitfalls of Comparison. An accurate comparison between a coprocessor supported simulation and a non-coprocessor supported simulation is difficult to achieve. The pitfalls in trying to make a fair comparison are discussed in the following sections.

5.8.1.1 Design Differences. The microcode synchronization routines and the SPECTRUM routines accomplish different design objectives. The microcode routines

were designed to provide a testbed to build a general-purpose architecture that could execute user-defined synchronization routines. The SPECTRUM routines were designed to provide a testbed for comparing different parallel simulation protocols in a common environment. Because of the different design objectives, the synchronization tasks of each implementation are different, and the performance of these synchronization tasks are difficult to compare.

5.8.1.2 Interdependence. Not all of the synchronization overhead has been off-loaded to the coprocessor. For example, the coprocessor implementation requires that the host format and transmit real messages. Furthermore, the coprocessor currently operates in the I/O space of the host; therefore, all synchronization tasks will require some work by the host. The interdependence between the host and the coprocessor makes it difficult to determine the true performance of the coprocessor.

5.8.1.3 Data Collection Overhead. The AFIT Algorithm Animation Research Facility (AAARF) was used to collect data on a parallel simulation executed on the Hypercube. The AAARF is documented in (5). The AAARF accurately measures performance within un-nested blocks of interest; however, the AAARF instrumentation requires that binary data be written to file once a temporary buffer in RAM has reached its capacity. This file I/O occurs outside of the measured blocks; therefore, the measured blocks are accurate. However, the total simulation time is distorted, and the amount of distortion is dependent on the size of the temporary buffer and the amount of data points that is collected.

Because of the AAARF overhead, interesting blocks could not be measured within the same execution of the simulation. Multiple iterations had to be executed to measure the different blocks of interest. Finally, measurements had to be taken with AAARF disabled to determine the correct application time.

5.8.2 Valid Comparisons. The coprocessor Post Message routine closely resembles the **LP_NQ_EVENT** routine in the SPECTRUM code. The only difference is that the coprocessor Post Message routine also updates the status of the input arcs. The equation for the coprocessor Post Message routine can be modified to compensate for this difference and is shown in Equation 6.

$$Post\ Msg = Fetch + 196(\%Write_R) + 208(\%Write_U) - 4(\%Null) \quad (6)$$

This modified equation is similar to Equation 3 except the time required to update the status of the input arc is not considered in the modified equation. This new equation allows for a fair comparison of the time required to insert events into the NEQ.

The coprocessor Get Event routine closely resembles the SPECTRUM Get Event routine. The major difference is that the coprocessor returns an error message if an event is not ready (thus ending the operation), and the SPECTRUM routine blocks until it is safe to proceed. Furthermore, the coprocessor Get Event routine updates the output arcs when a null message is retrieved, and the SPECTRUM routine updates the output arcs prior to the LP blocking (if the LP must block). Assuming that the difference in the way the output arc is updated is negligible, a comparison can be made between the two routines if the SPECTRUM block time can be accounted for.

The coprocessor Post Event routine cannot be compared with its counterpart in the SPECTRUM implementation. The coprocessor implementation only handles null messages which result from the real messages that are sent by the host. The SPECTRUM implementation handles both real and null messages.

5.8.3 Data.

5.8.3.1 Simulation Configuration. The performance of the SPECTRUM routines was measured by using a parallel VHDL simulation taken from (2). The application was a four-node, four-LP, randomly-partitioned Wallace Tree Multiplier. Although

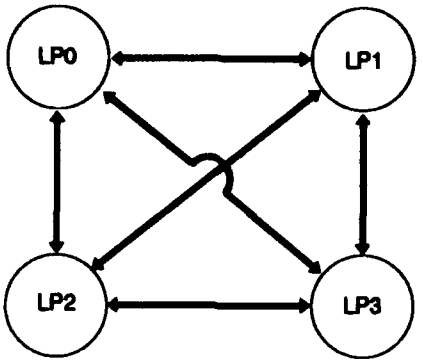


Figure 21. Simulation Configuration for Wallace Tree Parallel Simulation. Each LP is executed on a separate Hypercube node. Communication arcs are shown as bidirectional for clarity.

Table 7. *SPECTRUM NEQ Data. All quantities are mean values based on* data collected from 5 iterations of the Wallace Tree simulation. F_E is the fraction of the application that can be enhanced by accelerating the insertion of events into the NEQ. Application Time is measured with AAARF disabled.

LP	Avg Insert Time	#Calls	Total Insert Time	Application Time	F_E
LP0	128.6us	3907	502.44ms	25.67s	0.02
LP1	127.8us	4300	549.54ms	25.67s	0.02
LP2	130.2us	4819	627.43ms	25.48s	0.02
LP3	113.4us	5122	580.83ms	25.41s	0.02

the multiplier is a combinational circuit, feedback was introduced into the simulation because of the random partitioning strategy. The configuration of the simulation is shown in Figure 21.

5.8.3.2 AAARF Data. Table 7 shows the relevant measurements for the `LP_NQ_EVENT` procedure in SPECTRUM. The queue size never exceeded 10 messages, and the average insertion point was at the tail of a list of two messages. These observations are

Table 8. Coprocessor NEQ Data

LP	%Null	%Real	%Write$_R$	%Write$_U$
LP0	94	6	50	50
LP1	84	16	50	50
LP2	74	26	50	50
LP3	64	36	50	50

reflected in Table 7 by the small percentage of total application time that can be enhanced through acceleration of the `LP_NQ_EVENT` procedure.

The AAARF could not be used to collect reliable data on the Get Event function in SPECTRUM because of the strong interdependence of the LPs in the simulation. While LP_i is blocking and waiting for LP_j to send a message, LP_j is writing AAARF data out to file. The measured times were distorted so that the total block time was longer than the total Get Event time, and the total Get Event time was longer than the total simulation time. Because of this distortion, reliable data could not be collected on the SPECTRUM Get Event function.

5.8.3.3 Coprocessor Data. Table 8 shows the relevant data that is used for the coprocessor NEQ performance. The percentage of real and null messages was determined by using the `-DCOUNTS` option while running the Wallace Tree simulation on the Hypercube. The percentage of writes to reserved and unreserved ESAM words is based on the typically small NEQ size. As the size of the NEQ increases, the percentage of writes to unreserved words would increase.

The Coprocessor NEQ performance is shown in Table 9. The average insert time is based on Equation 6 and the data in Table 8. A clock period of 115ns is assumed as discussed in Section 5.6.

5.8.4 Partial Speedup Analysis. The speedup of the `LP_NQ_EVENT` procedure and the application are shown in Table 10. Equation 7 was used to calculate the speedup.

Table 9. Coprocessor NEQ Performance

LP	Avg Insert Time	#Calls	Total Insert Time
LP0	32.09us	3907	125.38ms
LP1	32.09us	4300	137.99ms
LP2	32.20us	4819	155.17ms
LP3	32.20us	5122	164.93ms

Table 10. Partial Speedup Analysis

LP	LP_NQ_EVENT	Application
LP0	4.00	1.02
LP1	3.98	1.02
LP2	4.04	1.02
LP3	3.52	1.01

$$Speedup = \frac{Execution\ Time_{OLD}}{Execution\ Time_{NEW}} = \frac{1}{(1 - F_E) + \frac{F_E}{Speedup_E}} \tag{7}$$

F_E is the fraction of the application that is enhanced as shown in Table 7, and $Speedup_E$ is the speedup of the enhanced portion of the application as shown in column 2 of Table 10. As expected, the application speedup is insignificant because of the small fraction of the application that can be enhanced.

5.9 Conclusion

A partial performance analysis of the coprocessor was done using the following data that was described in this chapter:

- Timing measurements of actual circuits.

- Coprocessor performance characteristic equations.
- Performance measurements extracted from a parallel simulation on a Hypercube.

The critical path of the coprocessor was determined to be the extreme search data path of the ESAM since the worst-case search for the minimum time-stamped event could require 460ns. Based on this time requirement and the use of a clock filter, the maximum operating frequency of the coprocessor was determined to be 8.7 MHz. The NEQ performance of the coprocessor was calculated by using characteristic equations for the coprocessor performance and data collected from a parallel Wallace Tree simulation. Although the coprocessor could provide a four-fold speedup for NEQ management, the total speedup was only 1.02 since less than 2% of the application was accelerated.

The results of this analysis were unfavorable towards the use of the Extreme Search Associative Memory for NEQ management because of the small fraction of the application that could be enhanced. Furthermore, no conclusions were made regarding the acceleration of other synchronization tasks because of the data collection overhead and interference with the simulation. Differences between the coprocessor synchronization tasks and the SPECTRUM synchronization tasks also prevented accurate conclusions regarding the acceleration of synchronization tasks.

The performance analysis in this chapter does not necessarily discredit the use of the ESAM for NEQ management. Only one simulation was used for performance measurements, and that simulation used random partitioning coupled with the conservative Chandy-Misra protocol. Based on these limited observations, the ESAM is not advantageous for synchronization-level NEQ management of randomly-partitioned, conservative parallel simulations. Other types of partitioning strategies and simulation protocols which allow for greater parallelism may produce more favorable results. Furthermore, application-level NEQ management may provide more favorable results than synchronization-level NEQ management.

VI. Conclusion

6.1 Introduction

This chapter explains the conclusions that can be made from this research and makes recommendations for follow-on research. The conclusions and recommendations cover the areas of NEQ acceleration and synchronization acceleration.

6.2 Conclusions

6.2.1 NEQ Acceleration. The Extreme Search Associative Memory was included in the DES Coprocessor to provide O(1) insert and retrieval time of events in the NEQ. The speedup that can be achieved by replacing a software NEQ with the ESAM is dependent on several factors:

1. The average size of the queue.
2. The efficiency of the software algorithm which maintains the queue.
3. The frequency of queue accesses.
4. The size and performance of the associative memory which is replacing the software queue.

The first three factors determine the fraction of the application that can be enhanced. For the Wallace Tree simulation that was analyzed in Chapter V, the software algorithm is a doubly-linked list which has only O(n) time efficiency for inserting events. This is generally an inefficient algorithm; however, the average size of the queue and the frequency of queue accesses were small enough that the efficiency of the algorithm was irrelevant.

The fourth factor in the above list determines the speedup of the fraction that can be enhanced. A 32-word associative memory was used for this research; therefore, this design would only be sufficient for applications which have queue sizes no larger than 32 words. Although the average queue size is small, speedup can still be achieved if the frequency of queue accesses is high. Furthermore, the size of the ESAM can be increased to allow for larger queue sizes; however, this increase will impact the performance of the memory.

Banton determined that an increase in the number of words causes a linear increase in the execution time of the extreme search (1:137).

Although a four-fold speedup was achieved for the NEQ management of the Wallace Tree simulation, the total speedup was only 1.02 since less than 2% of the application was accelerated. These results indicate that the ESAM is not advantageous for NEQ management of randomly-partitioned, conservative, parallel simulations. Other types of partitioning strategies and simulation protocols may yield different results. Furthermore, only synchronization-level NEQ management was considered. Application-level NEQ management may also yield different results.

6.2.2 Synchronization Acceleration. Acceleration of the synchronization routines is achieved by the use of specialized microcode-control hardware. Although the coprocessor performance can be characterized by equations, the actual speedup was not determined. The speedup that can be achieved by the coprocessor is dependent on several factors:

1. The fraction of simulation time that is used for parallel synchronization.
2. The performance of the microcode synchronization routines.
3. The amount of concurrency that can be achieved between the application and synchronization tasks.

The switching speed of the coprocessor will likely be less than that of the current commercial host processors because of the amount of resources used to optimize the design. Because of the longer propagation delays in the AFIT DES Coprocessor, the clock speed will also be less. For example, the results of this research indicate that the maximum operating frequency of the coprocessor will only be 8.7MHz. Although this clock speed is slow when compared to current commercial processors, the coprocessor should provide acceleration since the synchronization is executed in user-defined microcode.

Further potential for speedup exists when the synchronization routines can be executed concurrently with the application. In order for this potential to be realized, the coprocessor needs access to the same resources as the host. For example, if the coprocessor

has access to the multiprocessor communications resources, then the coprocessor can send and receive messages without involvement from the host. The host would then be free to concurrently execute the application while the coprocessor executes the synchronization.

6.3 Recommendations

6.3.1 NEQ Component. Future research concerning the ESAM and the NEQ Component should concentrate on the following topics:

- Identification of applications that can benefit from the design. Both parallel and sequential simulations should be explored. Synchronization-level and application-level NEQ management should be considered for parallel simulations.
- Fabrication of a single-ch , NEQ Component using the subcircuits which were validated in this research. The entire design can fit on a large-frame IC using a 2μm process, or a 1.2μm process can be used to achieve faster speeds.
- Design of an optimized memory array. The current design uses extreme search memory cells for all 32 bits of each word. Extreme searches are executed across the lower 17 bits, and equivalence searches are executed on the upper 15 bits. Circuit area and power consumption can be minimized by tailoring the memory cells to the type of associative operation that is required. Furthermore, SRAM cells could be added to the memory array to eliminate the need for a separate IC to store adjacent data.
- Scalability to support larger queues. Alternatives should be researched which minimize the impact on performance as the number of words in the array are increased. One option may be to use multiple ESAM arrays on the same IC.

6.3.2 Target Architecture. The DES Coprocessor was designed as a general-purpose I/O device to insure flexibility in choosing the target architecture. Once a target architecture is chosen, the coprocessor needs to be wire-wrapped on a board that meets the physical interface specifications. Additional logic design will also be required to insure

that the coprocessor meets the timing and control specification of the architecture. The i/PSC2 Hypercube has always been the target architecture for this research; however, other alternatives should be considered. The two primary alternatives can be classified by risk:

- A low-risk option is to interface the DES Coprocessor board to the VME bus prior to the Hypercube. The VME bus is well-documented, and all of the required hardware and test equipment is already co-located in the AFIT Advanced Digital Design Lab. The primary benefit of this approach is that the internal coprocessor architecture can be easily debugged. The primary cost of this approach is that the impact on parallel performance cannot be measured. Furthermore, the overhead of designing the VME interface will delay the progress of this research.
- A high-risk option is to interface the DES Coprocessor board to the Hypercube which uses a modified Multibus II architecture. The primary benefit of this approach is that meaningful performance comparisons can be made sooner. The primary cost of this option is the increased design complexity since both the internal coprocessor architecture and the interface must be resolved simultaneously.

A more detailed analysis needs to be conducted before committing to either option.

6.4 Summary

This research began with a detailed analysis of the initial DES Coprocessor design that was provided from previous research at AFIT. Based on that analysis, modifications to the design were required to make the coprocessor a viable circuit. Several of the coprocessor components were fabricated and tested for performance. Finally, a partial speedup analysis of the coprocessor was done using the timing data of actual circuits, characteristic performance equations of the coprocessor, and performance measurements from Hypercube parallel simulations.

Appendix A. Testbench-Coprocessor Interface

A.1 Interface Signals

Table 11 explains the signals which interface the DES Coprocessor to the VHDL testbench as shown in Figure 5. Table 12 is the truth table for data transfer between the host and coprocessor. Since the interrupt register, status register, and PARIO buffer all share the coprocessor bidirectional data port, simultaneous operations with these components will result in corrupted data. For example, if the status register is being read during an interrupt acknowledge, the status data and the interrupt vector will collide on the coprocessor data port.

A.2 Macroinstruction Set

A.2.1 Initialize Coprocessor. The register file and NEQ Component must be initialized prior to executing the first fetch/decode routine. After a system reset, the coprocessor waits for a 32-bit opcode which equals zero (decimal). This opcode starts the initialization of the register file and NEQ Component. Initialization operands are read from the host and written to the register file. The initial values of the registers are read from an ASCII file and are shown in Table 13. All of the initialized registers are dedicated to storing these values for the duration of the simulation. The only exception is Register 9 which holds the NEQ initialization value. Once the NEQ Component is initialized, this register can be used to store other values.

A.2.2 Initialize Simulation. The format for the Initialize Simulation instruction is shown in Table 14. The "To Node" and "To LP" fields identify the LP that is being initialized.

The format of the operands for this instruction is shown in Table 15. The fourth operand for this instruction specifies the Node/LP identifier for each I/O arc of the LP that is being initialized. The input arcs are specified before the output arcs.

Table 11. Coprocessor Interface Signals

Signal	Description
W_R	Specifies a write or read of status register or PARIO buffer in interface unit.
CS_DATA	Selects the PARIO buffer for write/read.
CS_STATUS	Selects the status register for write/read.
RESET	Resets the coprocessor.
CLK	Provides the master clk signal for coprocessor.
INTA	Interrupt acknowledge from host. Causes the interrupt vector to be output to data bus.
INTR	Interrupt request from the coprocessor.
DATA(31:0)	Bidirectional data transfer between the host and coprocessor.

Table 12. Truth Table of Coprocessor Interface Signals

W_R	CS_DATA	CS_STATUS	Operation
1	1	0	Write data to PARIO buffer.
0	1	0	Read data from PARIO buffer.
1	0	1	Write to status register.
0	0	1	Read status register.

Table 13. Register Initialization Data

Reg #	Value	Purpose
3	00000000000000000000000000000001	Maintain Arcs Status
5	00000000000000000000001111111111	Count Mask
6	00000011111111000000000000000000	To Node/LP Mask
7	00000000000000111111110000000000	From Node/LP Mask
9	00000000000000000000000000000000	NEQ Initialization

Table 14. Initialize Simulation Opcode Format

Bits	Opcode Field
31-26	Opcode Identifier (000100)
25-23	To Node
22-18	To LP
17-10	Unused
9-0	Count of Operands

Table 15. Initialize Simulation Operands

Operand Number	Bits	Field
1	31-0	LP Delay
2	31-0	Initial Sim Time
3	31-16	Number of LP Output Arcs
	15-0	Number of LP Input Arcs
4 until complete	31-18	Unused
	17-15	I/O Arc Node Identifier
	14-10	I/O Arc LP Identifier
	9-0	Unused

At the completion of this instruction, the coprocessor will output a null message on each output arc of the LP that is being initialized. The time stamp of the null message is the LP delay added to the initial simulation time. The coprocessor generates an interrupt to the host for each null message that is output.

A.2.3 Post Message. The format for the Post Message instruction is shown in Table 16. The format of the operands for this instruction is shown in Table 17. If the opcode specifies that the count of operands is two, then both operands are required and a real message is posted. If the opcode specifies that the count of operands is only one, then only the first operand is required and a null message is posted.

Table 16. Post Message Opcode Format

Bits	Opcode Field
31-26	Opcode Identifier (100000)
25-23	To Node
22-18	To LP
17-15	From Node
14-10	From LP
9-0	Count of Operands

Table 17. Post Message Operands

Operand Number	Bits	Field
1	31-17	Unused
	16-0	Time Tag
2	31-0	Memory Pointer

A.2.4 Get Event. The format for the Get Event instruction is shown in Table 18. The "To Node" and "To LP" fields identify the LP for which an event is being retrieved. No operands are used for this instruction. If a real message is retrieved, the coprocessor interrupts the host and provides the updated simulation time and the 32-bit memory pointer to the event data. If a null message is retrieved, the coprocessor processes null

Table 18. Get Event Opcode Format

Bits	Opcode Field
31-26	Opcode Identifier (010000)
25-23	To Node
22-18	To LP
17-0	Unused

Table 19. Post Event Opcode Format

Bits	Opcode Field
31-26	Opcode Identifier (001000)
25-23	To Node
22-18	To LP
17-15	From Node
14-10	From LP
9-0	Unused

messages for the output arcs of this LP. The coprocessor interrupts the host for each null message that is output.

A.2.5 Post Event. The format for the Post Event opcode is shown in Table 19. This instruction is used to process null messages for all of the output arcs of the specified LP except the output arc that received the real message. The host is responsible for formatting and sending the real message. The "To Node" and "To LP" fields specify the recipient of the real message. The "From Node" and "From LP" fields identify the sender of the real message. This opcode uses one operand to specify the time tag of the real message that was sent by the host. The coprocessor uses this time tag for each of the null messages that is processed. The coprocessor interrupts the host for each null message that is output.

A.3 Interrupt Vectors

The coprocessor uses interrupts to signal that the results of an operation are available. Three interrupt vectors are used (Table 20). The interrupt vectors are derived from the values of the dedicated registers which are loaded at initialization (Table 13).

The assertion of **INTA** from the host causes the 8-bit interrupt vector to be output on the coprocessor data port. Each interrupt has at least one operand that must be read from the PARIO buffer after the interrupt vector has been read. The format of the first

Table 20. Interrupt Vectors (decimal values)

Vector	Purpose
3	Null message from Post Event
254	Real Message from Get Event
255	Null Message from Init Sim or Get Event

Table 21. Interrupt Operand Format

Bits	Interrupt Operand Field
31-26	Unused
25-23	To Node
22-18	To LP
17-15	From Node
14-10	From LP
9-0	Count of Remaining Operands

interrupt operand is in Table 21. For null messages, the count of remaining operands is always one. This final operand is used to provide the time tag of the null message. For real messages, the count of remaining operands is always two. These two operands specify the time tag and memory pointer of the real message.

Table 22. Error Vectors (decimal values)

Vector	Purpose
1	ESAM full, Post Message failed
255	Unsafe to retrieve message for specified LP, Get Event failed

A.4 Error Vectors

The coprocessor uses error vectors to specify the type of error that occurred during an operation (Table 22). The error vectors are derived from the values of the dedicated registers which are loaded at initialization (Table 13). Errors are signaled through the status register, and error vectors are read from the PARIO buffer. Error vectors can also be used to specify an operand count mismatch or an illegal operand value; however, these error conditions were not tested.

A.5 Sample VHDL Simulation Input File

VHDL simulations of the coprocessor are executed by reading an ASCII instruction file. All of the instructions which are read from this file are in decimal format. This sample instruction file executes the following five instructions:

- Initialize Simulation for Node 2/LP 2.
- Post Message for Node 2/LP 2, from Node 0/LP 0.
- Post Message for Node 2/LP 2, from Node 1/LP 1.
- Post Message for Node 2/LP 2, from Node 2/LP 2.
- Get Event for Node 2/LP 2.

```
INSTR#  TO_NODE  TO_LP FROM_NODE FROM_LP COUNT
   4        2       2       0        0      9  ; Init Sim Opcode
OPERANDS
   5                                           ; LP Delay
   0                                           ; Initial LP Sim Time
   3       3                                   ; Count of Output/Input Arcs
   0       0                                   ; 1st Input Arc Identifier
   1       1                                   ; 2nd Input Arc Identifier
   2       2                                   ; 3rd Input Arc Identifier
   2       2                                   ; 1st Output Arc Identifier
   3       3                                   ; 2nd Output Arc Identifier
   4       4                                   ; 3rd Output Arc Identifier
INSTR#  TO_NODE  TO_LP FROM_NODE FROM_LP COUNT
  32       2       2       0        0       1  ; Post Message Opcode
OPERANDS
   5                                           ; Null Message Time Stamp
INSTR#  TO_NODE  TO_LP FROM_NODE FROM_LP COUNT
  32       2       2       1        1       2  ; Post Message Opcode
OPERANDS
   7                                           ; Real Message Time Stamp
  15                                           ; Real Message Mem Ptr
INSTR#  TO_NODE  TO_LP FROM_NODE FROM_LP COUNT
  32       2       2       2        2       2  ; Post Message Opcode
OPERANDS
   8                                           ; Real Message Time Stamp
  31                                           ; Real Message Mem Ptr
INSTR#  TO_NODE  TO_LP FROM_NODE FROM_LP COUNT
  16       2       2       0        0       0  ; Get Event Opcode
```

A.6 Sample VHDL Simulation Output File

Results from a VHDL simulation are formatted and output to an ASCII file. This section contains a sample output file. This output file is based on the input file of the previous section. The first three interrupts are generated from the Initialize Simulation instruction. The next three interrupts are generated from the Get Event instruction. Since a null message was retrieved during the Get Event, the coprocessor processes null messages for all of the output arcs of the LP.

```
-------------------------------------------------------------------------
INT VECTOR  TO_NODE  TO_LP  FROM_NODE  FROM_LP  PACKET COUNT   PACKET DATA
       255        2      2          2        2             1    5
-------------------------------------------------------------------------
INT VECTOR  TO_NODE  TO_LP  FROM_NODE  FROM_LP  PACKET COUNT   PACKET DATA
       255        3      3          2        2             1    5
-------------------------------------------------------------------------
INT VECTOR  TO_NODE  TO_LP  FROM_NODE  FROM_LP  PACKET COUNT   PACKET DATA
       255        4      4          2        2             1    5
-------------------------------------------------------------------------
INT VECTOR  TO_NODE  TO_LP  FROM_NODE  FROM_LP  PACKET COUNT   PACKET DATA
       255        2      2          2        2             1   10
-------------------------------------------------------------------------
INT VECTOR  TO_NODE  TO_LP  FROM_NODE  FROM_LP  PACKET COUNT   PACKET DATA
       255        3      3          2        2             1   10
-------------------------------------------------------------------------
INT VECTOR  TO_NODE  TO_LP  FROM_NODE  FROM_LP  PACKET COUNT   PACKET DATA
       255        4      4          2        2             1   10
-------------------------------------------------------------------------
```

Appendix B. NEQ Component

B.1 ESAM

The ESAM is documented in (1); however, several changes were made to the design. The most significant change is that all of the control signals have been changed.

Refer to Figure 9. All ESAM operations require a sequence of three control stimuli applied to the `CTRL(7:1)`, `CTRL0`, `ARRAY_CTRL(1:0)`, and `WRITE` ports. The VHDL model control signals required for each type of associative operation are shown in Table 23, and the control signals required for each type of I/O operation are shown in Table 24.

Read and write operations are executed based on the results of associative operations. A Write-Word or Read-Word can be executed consecutively until the results of the associative operation have been exhausted. Once the results of an associative operation have been exhausted, another associative operation must be executed prior to the next I/O operation.

B.2 NEQ Control Unit

Refer to Figure 7. The NEQ Control Unit controls the ESAM and the I/O port latches of the NEQ Component. The NEQ Control Unit has registered outputs in order to guarantee stable control signals. The following sections provide detailed information on the states shown in Figure 8. Table 25 shows the format of ESAM words which are used in these algorithms.

B.2.1 State Algorithms.

B.2.1.1 Init ESAM. The "Init ESAM" state initializes all of the words in the ESAM to a user-defined value and is executed as follows:

```
0. Signal not ready.
1. Latch the 32-bit initialization value from the coprocessor
   internal data path into the ESAM_DATA_IN register.
```

Table 23. ESAM Control Stimuli For Associative Operations

Operation	CTRL(7:1)	CTRL0	ARRAY_CTRL(1:0)	WRITE
Search-All	0101000	1	00	1
Equal	0111010	1	00	1
	0101000	1	00	1
Search-Subset	0101000	0	00	1
Equal	0111010	0	00	1
	0101000	0	00	1
Search-All	1101000	1	00	1
Not Equal	1111010	1	00	1
	1101000	1	00	1
Search-Subset	1101000	0	00	1
Not Equal	0111010	0	00	1
	1101000	0	00	1
Search-All	0101000	1	10	0
Minimum	0111010	1	10	0
	0101000	1	10	0
Search-Subset	0101000	0	10	0
Minimum	0111010	0	10	0
	0101000	0	10	0
Search-All	1101000	1	10	0
Not Minimum	1111010	1	10	0
	1101000	1	10	0
Search-Subset	1101000	0	10	0
Not Minimum	1111010	0	10	0
	1101000	0	10	0
Search-All	0101000	1	11	0
Maximum	0001110	1	11	0
	0101000	1	11	0
Search-Subset	0101000	0	11	0
Maximum	0111010	0	11	0
	0101000	0	11	0
Search-All	1101000	1	11	0
Not Maximum	1111010	1	11	0
	1101000	1	11	0
Search-Subset	1101000	0	11	0
Not Maximum	1111010	0	11	0
	1101000	0	11	0

Table 24. ESAM Control Stimuli For I/O Operations

Operation	CTRL(7:1)	CTRL0	ARRAY_CTRL(1:0)	WRITE
Write-All	0001100	1	00	1
	0001110	1	00	1
	0001100	1	00	1
Write-Subset	0001100	0	00	1
	0001110	0	00	1
	0001100	0	00	1
Write-Word	0000101	1	00	1
	0010111	1	00	1
	0000101	1	00	1
Read-Word	0000101	1	00	0
	0010111	1	00	0
	0000101	1	00	0

Table 25. Format of ESAM Word

Bits	Field	Controlled By
31	Valid Bit	NEQ Control Unit
30-26	To LP, message recipient	Host
25-23	From Node, message sender	Host
22-18	From LP, message sender	Host
17	Reserved Bit	NEQ Control Unit
16-0	Time tag of message	Host

```
2. Write-All. Valid Bit = 0, Reserve Bit = 0.
3. Search-All Equal for initialization value.
4. Signal ready.
```

B.2.1.2 Reserve Arcs. The "Reserve Arcs" state writes one word at a time to the ESAM. The write operation is based on the results of the Search-All Equal operation which ended the "Init ESAM" state. The Reserve Arcs state can be executed as many times as there are words in the ESAM. For example, if the ESAM has 32 words, then 32 Reserve Arcs can be executed. An attempt to reserve more than 32 arcs will result in an error condition.

This state executes the following algorithm:

```
0. Signal not ready.
1. Latch the 32-bit ESAM word from the coprocessor
   internal data path into the ESAM_DATA_IN register.
2. Write-Word. Valid Bit = 0, Reserve Bit = 1.
3. If word not written, signal error.
4. Signal ready.
```

B.2.1.3 Write Word. The "Write Word" state executes the following algorithm:

```
0. Signal not ready.
1. Latch the 32-bit ESAM word from the coprocessor
   internal data path into the ESAM_DATA_IN register.
2. Search-All Equal for invalid-reserved word for this arc.
3. If match, Write-Word (routine complete).
   Valid Bit = 1, Reserve Bit not written.
4. If not match, Search-All Equal for invalid-unreserved word.
5. If match, Write-Word (routine complete).
   Valid Bit = 1, Reserve Bit not written.
6. If not match, signal error (ESAM full).
7. Signal ready.
```

B.2.1.4 Find Min. The "Find Min" state executes the following algorithm:

```
0. Signal not ready.
1. Latch the 32-bit ESAM word from the coprocessor
   internal data path into the ESAM_DATA_IN register.
   This word is used to identify the LP for which a message
   is being retrieved.
2. Search-All Equal for valid messages for this LP.
   If not match, signal error (Event not ready).
3. Search-Subset Minimum to find the minimum valid time-tagged
   message for this LP.
4. Read-Word to read the minimum time tagged event.
   If there is more than one minimum time-tagged event,
   the word ''closest to the top'' is read.
5. Latch the ESAM word into the 32-bit ESAM_DATA_OUT register.
   Latch the encoded ESAM address into the 5-bit ESAM_ADDR_OUT register.
6. Search-All Equal for messages for this LP.
7. Search-Subset Minimum to find the minimum time-tagged
   message for this LP.
8. Write-Word to invalidate the minimum time-tagged message.
   Valid Bit = 0, Reserve Bit not written.
9. Signal ready.
```

B.2.1.5 Search. The "Search" state executes the following algorithm:

```
0. Signal not ready.
1. Latch the 32-bit ESAM word from the coprocessor
   internal data path into the ESAM_DATA_IN register.
   This word is used to identify the arc which is being searched.
2. Search-All Equal for valid messages for the specified arc.
3. If not match, signal error. (Status register must be updated).
4. Signal ready.
```

B.2.2 Detailed State Descriptions. Tables 26 through 31 provide the detailed state information for the Finite State Machine (FSM) which is included in the NEQ Control Unit. Table 32 provides the binary encoding of the state labels which are used in the state tables. The input columns of the state tables correspond respectively to the FSM_CTRL(2:0), MATCH_STAT, and WD_SEL_STAT ports as shown in Figure 7. The output columns correspond respectively to the ARRAY_CTRL(1), WRITE, CTRL(6:0), RD_DATA,

Table 26. Initialize ESAM States. "x" is a don't care value.

Inputs	Present State	Next State	Outputs
xx0xx	idle1	idle1	00000000000000xxxxxxx01
xx1xx	idle1	init1	01001100110000001111100
xxxxx	init1	init2	01001110100000001111100
xxxxx	init2	init3	01001100100000001111100
xxxxx	init3	init4	01101000100000001111100
xxxxx	init4	init5	01111010100000001111100
xxxxx	init5	reserve1	01101000100000001111100

Table 27. Reserve Arc States

Inputs	Present State	Next State	Outputs
x00xx	reserve1	reserve1	00000000000000xxxxxxx01
x01xx	reserve1	reserve2	01000101110000010111000
xxxxx	reserve2	reserve3	01010111100000010111000
xxx01	reserve3	reserve4	01010111100000010111000
xxxxx	reserve4	reserve1	01000101100000010111000
xxx00	reserve3	error1	01000101100000010111000
xx0xx	error1	error1	00000000000000xxxxxxx10
xx1xx	error1	idle1	00000000000000xxxxxxx00

WR_DATA, **OE_DATA**, **WR_ADDR**, **OE_ADDR**, **DATA_IN(31)**, **DATA_IN(17)**, **MASK(5:0)**, **ERROR**, and **READY** ports.

The mask field which is output from the Finite State Machine (FSM) is only 5 bits; however, the NEQ Control Unit outputs a 6-bit mask field. Each of the mask bits which are output from the FSM correspond to the ESAM-word fields which are listed in Table 25. The most-significant mask bit corresponds to the valid-bit field, and the least significant mask bit corresponds to the time-tag field. The least-significant bit of the mask field from the FSM drives the two least-significant bits from the NEQ Control Unit. This fanout was

Table 28. Write Word States

Inputs	Present State	Next State	Outputs
x10xx	reserve1	write1	011010001100000111111000
xxxxx	write1	write2	011110101000000111111000
xxx1x	write2	write3	011010001000000111111000
xxxxx	write3	write4	010001011000001011110100
xxxxx	write4	write5	010101111000001011110100
xxxxx	write5	write6	010101111000101011110100
xxxxx	write6	idle2	010001011000001011110100
xxx0x	write2	write7_0	011010001000000001001000
xxxxx	write7_0	write7_1	011010001000000001001000
xxxxx	write7_1	write8	011110101000000001001000
xxx1x	write8	write3	011010001000000001001000
xxx0x	write8	error2	011010001000000001001000
010xx	idle2	write1	011010001100000111111000

Table 29. Search States

Inputs	Present State	Next State	Outputs
001xx	idle2	search1	011010001100000101110000
xxxxx	search1	search2	011010100000101110000
xxx10	search2	idle2	010001000000101110000
xxx00	search2	error2	011010001000000101110000

introduced into the NEQ Control Unit design in order to reduce the load on the time-tag mask bit.

Table 30. Find Minimum States

Inputs	Present State	Next State	Outputs
011xx	idle2	min1	011010001100001011000000
xxxxx	min1	min2	011110101000001011000000
xxx0x	min2	error2	011010001000001011000000
xxx1x	min2	min3	011010001000001011000000
xxxxx	min3	min4	101010000000000000000100
xxxxx	min4	min5	101110100000000000000100
xxxxx	min5	min6	101010000000000000000100
xxxxx	min6	min7	000001011000000011111100
xxxxx	min7	min8	000101111000000011111100
xxxxx	min8	min9	000101111010100011111100
xxxxx	min9	min10	000001011000000011111100
xxxxx	min10	min11	011010001000001011000000
xxxxx	min11	min12	011110101000001011000000
xxxxx	min12	min13	011010001000001011000000
xxxxx	min13	min14	101010000000000000000100
xxxxx	min14	min15	101110100000000000000100
xxxxx	min15	min16	101010000000000000000100
xxxxx	min16	min17	010001011000000001000000
xxxxx	min17	min18	010101111000000001000000
xxxxx	min18	idle2	010001011000000001000000

Table 31. Idle and Error States

Inputs	Present State	Next State	Outputs
000xx	idle2	idle2	00000000000000xxxxxxx01
100xx	idle2	idle2	00000000000100xxxxxxx01
101xx	idle2	idle2	00000000000001xxxxxxx01
xx0xx	error2	error2	00000000000000xxxxxxx10
xx1xx	error2	idle2	00000000000000xxxxxxx01

Table 32. State Encoding

Label	Encoding	Label	Encoding	Label	Encoding
idle1	001110	init1	010000	init2	010100
init3	000000	init4	000001	init5	000100
reserve1	000101	reserve2	011101	reserve3	001101
reserve4	001100	error1	111111	write1	010101
write2	010111	write3	001010	write4	011111
write5	001001	write6	001111	idle2	000111
write7_0	100110	write7_1	010010	write8	000110
error2	101111	search1	010001	search2	000011
min1	010011	min2	000010	min3	100101
min4	100011	min5	100000	min6	011100
min7	011000	min8	011001	min9	001000
min10	010110	min11	001011	min12	100111
min13	100010	min14	100001	min15	100100
min16	011010	min17	011011	min18	011110

Appendix C. Microcode Control Engine

C.1 Introduction

This appendix provides a detailed description of the components of the Microcode Control Engine. Refer to Figure 13 for the Execution Unit schematic and Figure 14 for the Control Unit schematic.

C.2 Execution Unit

C.2.1 Register File. The **GPR_File** outputs the contents of the registers which are selected by the register decoders. The **R1_DECODER** selects one of sixteen registers, and the contents of the selected register are output on **A(31:0)**. Likewise, the **R2_DECODER** selects one of sixteen registers, and the contents of the selected register are output on **B(31:0)**. The **R1_DECODER** also selects a register for write operations. When **CLK4** and **AND_CTRL** are asserted, the output of the shifter is written to the selected register.

C.2.2 ALU. The ALU supports eight operations as shown in Table 33. The **Z_OUT** signal is high when the result of the ALU operation is zero. The **N_OUT** signal is high when the most significant bit of the ALU result is high. Two's complement notation is used.

Table 33. ALU Operations

Control Signal	Operation
000	Add
001	Increment
010	XOR
011	OR
100	Subtract
101	AND
110	Decrement
111	Pass-B

Table 34. Shifter Operations

Control Signal	Operation
000	No Shift
001	Shift Left One Bit
010	Shift Right One Bit
011	Shift Left Eight Bits
100	Shift Right Eight Bits
101-111	No Shift

Table 35. MBR Operations

Control Signal	Operation
00	No-op
01	Load data from shifter and output data
10	Read data from coprocessor internal data path
11	Output data

C.2.3 Shifter. The shifter supports five operations as shown in Table 34. The shifted-out data is not retained, and the shifted-in data is all zeroes.

C.2.4 MBR. The Memory Buffer Register supports four operations as shown in Table 35.

C.3 Control Unit

C.3.1 Microinstruction Decode Unit. This component decodes the 6-bit microinstructions which are discussed in Appendix D. The decoded instruction is latched on `CLK1`. Registered output was necessary in order to guarantee stable control signals.

Table 36. MSL Operations

Control Signal	Operation
000	if Negative jump
001	if ESAM Error jump
010	if Zero jump
011	if not Status(1) jump
100	if Status(0) jump
101	jump
110	if not ESAM Ready jump
111	no jump (select incrementer)

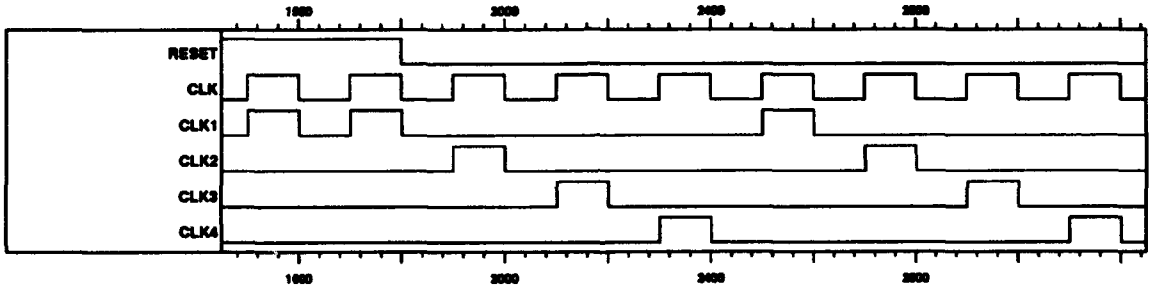


Figure 22. Microcode Control Engine Control Clock Waveform

C.3.2 Micro-sequencing Logic. This component determines the address of the next microinstruction in accordance with Table 36. The jump address and the incremented address are multiplexed through the Mapping Mux (**MMUX**).

C.3.3 Flag Register. This component latches the results of an ALU instruction when **CLK3** and **CTRL** are asserted.

C.3.4 Clock. This component is used to generate a four-phased non-overlapping clock signal to prevent race conditions in the microcode control engine (Figure 22). The

CLK1 signal pulses with the master clock signal for the duration of a reset. Once the reset is complete, the clock begins normal operation.

C.3.5 Micro-program Counter. This component latches the address of the next microinstruction on the rising edge of CLK4. The CLK1 port is used for a synchronous reset. The MPC is reset on the rising edge of CLK1 when the reset signal is asserted. If the reset signal does not overlap with the rising edge of CLK1, the MPC will not be reset.

Appendix D. DES Coprocessor Microcode

D.1 Microinstruction Set

This DES Coprocessor has 64 microinstructions (Table 37). The following rules are used for these microinstructions:

- For microinstructions with two register operands, the first operand is also the destination register.
- Only microinstructions which operate on registers affect the NZ flag register.
- Instructions eight through fifteen affect the NZ flag but the arithmetic result is not written to a register.
- Instructions sixteen through nineteen shift R2 and store the results in R1. In order to shift the contents of a register and store the results in the same register, R1 and R2 must address the same register.
- For Adjacent RAM operations, the address is specified from the NEQ Component. For SRAM operations, the address is specified from the Microcode Control Engine. For both operations, data is read and written through the MBR of the Microcode Control Engine.

D.2 Microcode Algorithms

D.2.1 Coprocessor Initialization.

```
1. Signal ready to host.
2. Wait for initialization opcode from host.
3. Wait for initialization operand from host.
4. Write initialization operand to register file.
5. Repeat 3 and 4 until complete.
6. Initialize NEQ Component.
7. Goto Fetch-Decode Routine
```

Table 37. DES Coprocessor Microinstruction Set (Decimal Base)

Instr #	Operation	Instr #	Operation
0	Adj RAM Write	32	mov(MAR,R1)
1	and(R1,R2)	33	NEQ Output Addr
2	xor(R1,R2)	34	lshift8(or(R1,R2))
3	or(R1,R2)	35	Interrupt Request
4	add(R1,R2)	36	SRAM Read
5	sub(R1,R2)	37	SRAM Write
6	incr(R1)	38	NEQ Init
7	decr(R1)	39	NEQ Clear Error
8	nz:=R1 and R2	40	NEQ Reserve Arc
9	nz:=R1 xor R2	41	NEQ Output Addr; Output MBR
10	nz:=R1 or R2	42	NEQ Read
11	nz:=R1 + R2	43	NEQ Write
12	nz:=R1 - R2	44	NEQ Find Min
13	nz:=R1 + 1	45	NEQ Search
14	nz:=R1 - 1	46	mov(MBR,PARIO)
15	nz:= R2	47	mov(PARIO,MBR)
16	lshift(R1,R2)	48	mov(MAR,R1); mov(MBR,R2)
17	rshift(R1,R2)	49	mov(R1,MBR)
18	lshift8(R1,R2)	50	mov(MBR,R2)
19	rshift8(R1,R2)	51	mov(R1,R2)
20	lshift(and(R1,R2))	52	toggle(status(2),status(0)
21	lshift(xor(R1,R2))	53	toggle(status(3))
22	lshift(or(R1,R2))	54	toggle(status(2))
23	lshift(add(R1,R2))	55	toggle(status(1))
24	lshift(sub(R1,R2))	56	toggle(status(0))
25	lshift(incr(R1))	57	if zero jmp R1/R2
26	lshift(decr(R1))	58	if neg jmp R1/R2
27	rshift(and(R1,R2))	59	if (status(1)=0) jmp R1/R2
28	rshift(xor(R1,R2))	60	if (status(0)=1) jmp R1/R2
29	rshift(or(R1,R2))	61	if NEQ not complete jmp R1/R2
30	rshift(add(R1,R2))	62	if NEQ error jmp R1/R2
31	Adj RAM Read	63	jmp R1/R2

D.2.2 Fetch-Decode.

```
1. Wait for opcode from host.
2. Signal not ready to host.
3. If NEQ Component error, clear NEQ Component Error.
4. R2 := opcode
5. R8 := "Count of Operands" field from opcode.
6. R9 := "To Node/LP" field from opcode.
7. R0 := LP SRAM partition pointer,
         (decoded from "To Node/LP" field of opcode).
8. Decode opcode field and branch to correct microroutine.
```

D.2.3 Initialize Simulation.

```
1.  Read LP Delay from host; R10 := LP Delay.
2.  Read Sim Time from host; R11 := Sim Time.
3.  Read # Arcs Out/In from host; R12 := # Arcs Out/In.
4.  Format LP input arcs status word.
5.  Write LP input arcs status word to SRAM.
6.  Write LP Delay to SRAM.
7.  Write Sim Time to SRAM.
8.  Write # Arcs Out/In to SRAM.
9.  Read input arc identifier from host.
10. Write input arc identifier to SRAM and reserve arc in NEQ.
11. Repeat 9-10 for count of input arcs.
12. Read output arc identifier from host.
13. Write output arc identifier to SRAM.
14. Repeat 12-13 for count of output arcs.
15. Send null messages on all output arcs.
16. Signal ready to host and jump to start of Fetch-Decode.
```

D.2.4 Post Message.

```
1. Read time tag from host; R9 := time tag.
2. If real message, read memory pointer from host; R10 := memory pointer.
   If null message, R10 := all 0's.
3. Format message for NEQ Component.
4. Write message to NEQ Component.
5. If NEQ Error, signal ESAM full error (exit this routine).
6. Write memory pointer to Adjacent SRAM.
7. Update LP input arcs status word.
8. Signal ready to host and jump to start of Fetch-Decode.
```

D.2.5 Get Event.

```
1. Check if valid events on all input arcs.
2. If not all input arcs have a valid event,
   signal error (exit this routine).
3. Find minimum event in NEQ Component.
4. Read minimum event from NEQ Component.
5. Read memory pointer from Adjacent SRAM.
6. Update LP Simulation time in SRAM.
7. Check if a null message was retrieved (mem ptr equal all 0's).
   a. If null message retrieved, output null messages on all output arcs.
   b. If real message retrieved, output updated sim time and memory pointer.
8. Signal ready to host and jump to start of Fetch-Decode.
```

D.2.6 Post Event.

```
1. Decode SRAM pointer from "From Node/LP" field of opcode; R0 := SRAM ptr.
2. Read Sim Time from host; R10 := Sim Time.
3. Read an output arc identifier from LP SRAM partition.
4. Compare output arc identifier with "To Node/LP" field of opcode.
   a. If not equal, send null message on output arc.
   b. If equal, do not send null message (arc that got the real message).
5. Repeat 3-4 for all output arcs of the LP.
6. Signal ready to host and jump to start of Fetch-Decode.
```

D.3 Microcode Preprocessor

D.3.1 Preprocessor Rules. The microcode preprocessor is used to convert the microcode input file into the correct format to be used in the behavioral description of the control store EPROM. The preprocessor is written in VHDL. The following fields are used in the microcode input file:

- `Instr` - This field gives the decimal value of the microinstruction.
- `# Reg` - This field gives the remaining number of fields (0, 1, or 2).
- `R1` - This field gives the R1 encoded value for the microinstruction.
- `R2` - This field gives the R2 encoded value for the microinstruction.
- `REL JMP` - This field gives the relative branch address for the microinstruction.

The following rules must be followed when writing or modifying the microcode input file:

- Full-line and end-of-line comments are permitted.
- Comments can begin with any non-numerical character.
- Branch targets are relative to the branch instruction.
- Full-line comments are not counted in relative branches.
- The microinstruction fields are decimal-based.

D.3.2 Sample Microcode Input File. This section shows the fetch-decode portion of the microcode input file. The actual input file has an additional "Comment" field after the "Pseudocode field." The comment field is not shown because a wide-carriage printer would be required.

```
*********************************************************************
*  This is the fetch/decode routine.
*********************************************************************
Instr  #Reg R1  R2   REL JMP     * Pseudocode
*********************************************************************
   59    1              0          if (Status(1) = 0) jmp 0
   53    0                         toggle Status(3)
   62    1              2          if NEQ_error jmp 2
   63    1              2          jmp 2
   39    1              0          NEQ_clear_error
   51    2   8   5                 mov(R8,R5)
   51    2   9   6                 mov(R9,R6)
   46    0                         mov(MBR,Pario_Data_In)
   49    2   2   0                 mov(R2,MBR)
    1    2   8   2                 and(R8,R2)
    1    2   9   2                 and(R9,R2)
   55    0                         toggle Status(1)
   51    2   0   9                 mov(R0,R9)
   19    2   0   0                 rshift8(R0)
   19    2   0   0                 rshift8(R0)
   16    2   0   0                 lshift(R0)
   16    2   0   0                 lshift(R0)
   16    2   0   0                 lshift(R0)
   51    2  10   2                 mov(R10, R2)
   58    1            123          if negative jmp 123
   16    2  10  10                 lshift(R10)
   15    2   0  10                 nz := R10
   58    1            241          if negative jmp 241
   16    2  10  10                 lshift(R10)
   15    2   0  10                 nz := R10
   58    1            191          if negative jmp 191
*********************************************************************
```

Appendix E. SRAM Memory Map

The SRAM memory map for the DES Coprocessor is shown in Figure 23. The SRAM is used by the Microcode Control Engine to store LP specific information which is required to synchronize the simulation. Furthermore, the SRAM is used by the NEQ Component to store memory pointers. These memory pointers provide the addresses to the next-event data which is stored in the main memory of the host.

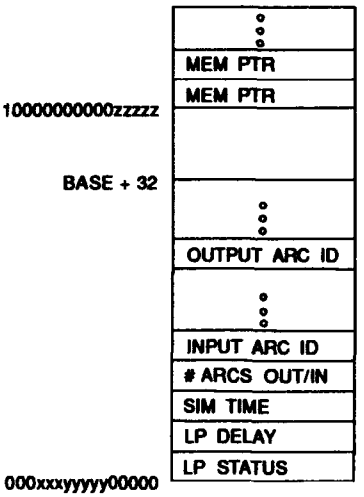


Figure 23. SRAM Memory Map for DES Coprocessor

Each LP is allowed a partition of 32 words in the low address space of the SRAM. The base pointer of the LP partition is calculated from the Node/LP identification. In Figure 23, the **xxx** and **yyyyy** bits of the 16-bit base address correspond to the "To Node" and "To LP" fields of the opcodes that are discussed in Appendix A.

Each ESAM word is mapped to one SRAM word. The **zzzzz** bits of the ESAM address space correspond to the 5-bit encoded ESAM address.

Appendix F. DES Coprocessor Interface Unit

F.1 Status Register

The Status Register in the DES Coprocessor Interface Unit stores a 4-bit value. The significance of the status bits is shown in Table 38.

Refer to Figure 15. In order to toggle a particular bit, the HOST_TOGGLE or COPROCESSOR_TOGGLE port is asserted, and a "1" is placed on the data line that corresponds to the bit that is to be toggled. A "0" signifies that the corresponding bit is not to be toggled. The Status Register is tied to the lowest four bits on the coprocessor data port to the host, and the lower two bits are used as feedback for the Microcode Control Engine.

The host waits until the Ready Bit is set and the Data-In Bit is clear before writing an Opcode to the Pario Buffer; however, the host only waits on a clear Data-In Bit before writing operands to the buffer.

The host toggles Bit 1 high after writing data to the Pario Buffer, and the coprocessor toggles the bit low after reading the data. The coprocessor toggles Bit 0 high after writing data to the Pario Buffer, and the host toggles the bit low after reading the data.

A clock filter synchronizes the host and coprocessor interface to the status register. Since four clock inputs to the coprocessor correspond to a full cycle of the four-phased non-

Table 38. Interface Unit Status Register

Bit	Use	Value	Toggled By
3	Ready Bit	1 = Ready 0 = Not Ready	Coprocessor
2	Error Bit	1 = Error 0 = No Error	Coprocessor
1	Data-In Bit	1 = Data-In Buffer Full 0 = Data-In Buffer Empty	Coprocessor and Host
0	Data-Out Bit	1 = Data-Out Buffer Full 0 = Data-Out Buffer Empty	Coprocessor and Host

overlapping clock of the control engine, the host is allowed to modify the Status Register only once every four input clock pulses.

F.2 Pario Buffer

The Pario Buffer is composed of two 32-bit registers. One register is a data input buffer, the other register is a data output buffer. The host writes data to the input buffer and reads data from the output buffer. The coprocessor writes data to the output buffer and reads data from the input buffer. Bits 1 and 0 of the Status Register are used to prevent data from being over-written before the recipient has read the data.

F.3 Interrupt Register

The Interrupt Register stores an 8-bit interrupt vector. When the coprocessor asserts **INTR**, the interrupt vector is latched into the register from the lower eight bits of internal coprocessor data path. When the host asserts **INTA**, the interrupt vector is output on the lower eight bits of the coprocessor data port.

Table 39. Pin Assignments for 132-Pin PGA ESAM Array. The following abbreviations are used for pin names: s = search select, w = word select, m = word match, di = data in, do = data out, v = vdd, g = gnd, nc = no connection.

Pin	Use	Pin	Use	Pin	Use	Pin	Use	Pin	Use
N3	s16	P5	w16	K3	di8	M2	mask0	N11	do18
H1	s15	C2	w15	L2	di7	M6	ctrl0	M10	do17
H2	s14	C3	w14	M1	di6	N6	ctrl1	N12	do16
H3	s13	B2	w13	K2	di5	N5	write	M11	do15
G3	s12	C4	w12	L1	di4	P7	do31	N13	do14
G2	s11	B3	w11	K1	di3	N7	do30	M12	do13
G1	s10	C5	w10	J3	di2	M7	do29	M13	do12
F1	s9	B4	w9	J2	di1	M8	do28	L12	do11
F2	s8	A3	w8	J1	di0	N8	do27	M14	do10
F3	s7	B5	w7	M5	mask8	P8	do26	L13	do9
E1	s6	A4	w6	P4	mask7	P9	do25	L14	do8
E2	s5	A5	w5	N4	mask6	N9	do24	K12	do7
E3	s4	C6	w4	P3	mask5	M9	do23	K13	do6
D1	s3	B6	w3	M4	mask4	P10	do22	K14	do5
D2	s2	A6	w2	M3	mask3	P11	do21	J12	do4
C1	s1	A7	w1	N2	mask2	N10	do20	J13	do3
D3	s0	B7	w0	L3	mask1	P12	do19	J14	do2
H14	do1	B10	m23	C13	m13	G14	m3	B1	g1
H13	do0	C10	m22	E12	m12	G13	m2	A13	g2
C7	m31	A11	m21	D13	m11	G12	m1	B14	g3
C8	m30	B11	m20	C14	m10	H12	m0	N14	g4
B8	m29	A12	m19	E13	m9	P1	v1	P2	g5
A8	m28	C11	m18	D14	m8	A1	v2	N1	g6
A9	m27	B12	m17	E14	m7	A2	v3	P6	nc
B9	m26	C12	m16	F12	m6	A14	v4		
C9	m25	B13	m15	F13	m5	P14	v5		
A10	m24	D12	m14	F14	m4	P13	v6		

Table 40. Mapping of Pin Inputs to Memory Inputs for 132-Pin PGA ESAM Array. In order to fit the chip on a 132-pin PGA, pin inputs were hard-wired to multiple memory inputs. This wiring scheme was used for the data, mask, search select, and word select inputs. The following abbreviations are used for pin names: s = search select, w = word select, di = data in.

Pin Input	Memory Input	Pin Input	Memory Input	Pin Input	Memory Input
mask8	mask31	mask7	mask30..28	mask6	mask27..24
mask5	mask23..20	mask4	mask19..16	mask3	mask15..12
mask2	mask11..8	mask1	mask7..4	mask0	mask3..0
di8	di31	di7	di30..28	di6	di27..24
di5	di23..20	di4	di19..16	di3	di15..12
di2	di11..8	di1	di7..4	di0	di3..0
s16/w16	word 31	s15/w15	word 30	s14/w14	word 29..28
s13/w13	word 27..26	s12/w12	word 25..24	s11/w11	word 23..22
s10/w10	word 21..20	s9/w9	word 19..18	s8/w8	word 17..16
s7/w7	word 15..14	s6/w6	word 13..12	s5/w5	word 11..10
s4/w4	word 9..8	s3/w3	word 7..6	s2/w2	word 5..4
s1/w1	word 3..2	s0/w0	word 1..0		

Table 41. Pin Assignments for 132-Pin PGA Word Select Circuit. The following abbreviations are used for pin names: s = search select, w = word select, m = word match, c = control, v = vdd, g = gnd, nc = no connection.

Pin	Use	Pin	Use	Pin	Use	Pin	Use	Pin	Use
N2	m0	D3	m27	M11	w14	N9	s9	A2	v3
L3	m1	C2	m28	M13	w15	P10	s10	A14	v4
M2	m2	C3	m29	M14	w16	N10	s11	P14	v5
K3	m3	B2	m30	L14	w17	N11	s12	P13	v6
L2	m4	C4	m31	K13	w18	N12	s13	B1	g1
M1	m5	B3	c7	J12	w19	M12	s14	A13	g2
K2	m6	C5	c6	J14	w20	L12	s15	B14	g3
L1	m7	B4	c5	H13	w21	L13	s16	N14	g4
K1	m8	B5	c4	G12	w22	K12	s17	P2	g5
J3	m9	A4	c0	G14	w23	K14	s18	N1	g6
J2	m10	C6	c3	F13	w24	J13	s19	N13	nc
J1	m11	A5	c2	E14	w25	H14	s20	C14	nc
H1	m12	A3	c1	E13	w26	H12	s21	A10	nc
H2	m13	M3	w0	E12	w27	G13	s22	C9	nc
H3	m14	M4	w1	D12	w28	F14	s23	B9	nc
G3	m15	N4	w2	C12	w29	F12	s24	A9	nc
G2	m16	M5	w3	C11	w30	D14	s25	A8	nc
G1	m17	P5	w4	B11	w31	D13	s26	B8	nc
F1	m18	N6	w5	N3	s0	C13	s27	C8	nc
F2	m19	P7	w6	P3	s1	B13	s28	C7	nc
F3	m20	M7	w7	P4	s2	B12	s29	B7	nc
E1	m21	N8	w8	N5	s3	A12	s30	A7	nc
E2	m22	P9	w9	M6	s4	A11	s31	A6	nc
E3	m23	M9	w10	P6	s5	B10	c_out	B6	nc
D1	m24	P11	w11	N7	s6	C10	flag		
D2	m25	P12	w12	M8	s7	P1	v1		
C1	m26	M10	w13	P8	s8	A1	v2		

Table 42. Pin Assignments for 40-Pin DIP NEQ Control Unit. The following abbreviations are used for pin names: fsm = fsm control, m = mask, ctrl = ESAM control, v = vdd, g = gnd, nc = no connection.

Pin	Use	Pin	Use	Pin	Use	Pin	Use	Pin	Use
27	fsm2	9	ctrl5	40	v_bit	21	rd_d	35	v2
31	fsm1	14	ctrl4	39	r_bit	19	wr_d	5	g1
29	fsm0	7	ctrl3	37	m5	23	oe_d	25	g2
32	wdsel	11	ctrl2	38	m4	18	wr_a	8	nc
33	match	6	ctrl1	36	m3	22	oe_a	16	nc
26	clock	12	ctrl0	3	m2	24	rdy	20	nc
30	reset	10	write	1	m1	17	error	28	nc
13	ctrl6	2	array	34	m0	15	v1	4	nc

Table 43. Pin Assignments for 40-Pin DIP Corrected ESAM Array. The following abbreviations are used for pin names: din = data in, wd = word select, srch = search select, mat = word match, dout = data out, nc = no connection.

Pin	Use	Pin	Use	Pin	Use	Pin	Use	Pin	Use
30	din5	36	mask3	23	ctrl1	20	dout5	12	mat1
32	din4	38	mask2	22	ctrl0	19	dout4	13	mat0
34	din3	40	mask1	3	write	18	dout3	15	vdd1
37	din2	2	mask0	4	prech	17	dout2	35	vdd2
39	din1	29	srch3	6	wd3	16	dout1	5	gnd1
1	din0	28	srch2	7	wd2	14	dout0	25	gnd2
31	mask5	27	srch1	8	wd1	10	mat3	24	nc
33	mask4	26	srch0	9	wd0	11	mat2	21	nc

Table 44. Pin Assignments for 84-Pin PGA Execution Unit. The following abbreviations are used for pin names: a = address, d = bidirectional data, nc = no connection.

Pin	Use	Pin	Use	Pin	Use	Pin	Use	Pin	Use
L1	vdd	J2	mbr1	K1	mbr0	J1	d3	H2	a2
H1	a4	G3	a5	G2	a6	G1	a3	F1	a11
F3	a1	E3	vdd	E1	a10	E2	a8	F2	a7
D1	a12	D2	d5	C1	d4	B1	gnd	C2	d6
B2	d11	A1	vdd	B3	d15	A2	d0	A3	d12
B4	d13	A4	d7	A6	d24	B5	d10	A5	d8
C5	d9	C6	d14	B6	a0	A7	d28	B7	d2
C7	nout	A8	zout	B8	d1	A9	d31	A10	gnd
B9	d26	B10	d27	A11	vdd	C10	d19	B11	d23
C11	d16	D10	d22	D11	d20	F11	d25	E10	d17
E11	d29	E9	d21	F9	d18	F10	a13	G11	a14
G10	a9	G9	a15	H11	d30	H10	nc	J11	nc
K11	gnd	J10	bmux	K10	mar_oe	L11	vdd	K9	alu1
L10	alu2	L9	alu0	K8	clk4	L8	clk2	J7	and
K7	shift0	L7	shift1	L6	shift2	J6	r2(0)	J5	r2(3)
L5	r1(2)	K5	r1(1)	K6	r2(2)	L4	r2(1)	K4	mar_ctrl
L3	r1(3)	L2	gnd	K3	clk3	K2	r1(0)		

Appendix H. Testing of Integrated Circuits

H.1 Introduction

This appendix explains the testing of the integrated circuits that were fabricated. The testing for the initial ESAM Array, ESAM Word Select Circuit, NEQ Control Unit, and corrected ESAM Array are discussed. Unless otherwise stated, all vectors are hexadecimal.

H.2 ESAM Array

Three separate tests were created for the 32-word by 32-bit ESAM Array. These tests were designed to:

- Validate the execution of associative operations.
- Measure the worst-case timing performance of associative operations.
- Validate the execution of I/O operations.

H.2.1 Validation of Associative Operations. The following test was used to validate the functionality of the associative operations:

```
1. Write 00000000 to all words (31 down to 0).
2. Write FFFFFFFF to words 15 down to 8.
3. Write 0F0F0F0F to words 23 down to 16.
4. Search-All maximum; match on words 15 down to 8.
5. Search-Subset maximum (subset = 23-16 and 7-0), match 23 down to 16.
6. Search-All minimum; match on words 31 down to 24 and words 7 down to 0.
7. Search-Subset minimum (subset = 15-0), match on words 7 down to 0.
8. Search-All equal to 00000000; match on words 15 down to 8.
```

All bits of the mask input to the memory were set to a high logic value so that all bit slices were tested. The equivalence search operation in step 8 shows that this function was not returning correct results. The Search-All equal operation should have returned matches on words 31 down to 24 and words 7 down to 0. Since the comparand data path was stuck at a high logic value, the operation incorrectly returned matches on words 15 down to 8.

H.2.2 Worst-Case Performance of Associative Operations. The following test was used to measure the worst-case performance of the extreme-search associative operations. The equivalence search was not evaluated in this test since this search operation failed the functional test.

```
1. Write FFFFFFFF to all words.
2. Write 00000000 to word 31.
3. Search-All minimum; match on word 31.
4. Write 00000000 to all words.
5. Write FFFFFFFF to word 31.
6. Search-All maximum; match on word 31.
7. Goto 1.
```

By running the test in an infinite loop, the polling time for the expected result (match) could be lowered until errors were detected. The minimum polling time at which errors were not detected was the minimum execution time of the search operation. The Search-All minimum typically failed before the Search-All maximum. When one of the extreme searches began to fail before the other, the expected result from the failing search was no longer polled. This technique allowed the polling time to be decreased until the other extreme search began to generate errors. Thus, the same test was used to measure the performance of both the maximum and minimum searches.

The above test was executed across search fields of 32, 31, 28, 24, 20, 16, 12, 8, and 4 bits. The size of the search field was controlled by setting the mask input to the desired value.

H.2.3 Validation of I/O Operations. The following test was used to validate the execution of I/O operations. The majority of the ICs (81%) failed this test since single-word reads were stuck at a high logic value.

```
1. Write FFFFFFFF to word x.
2. Read FFFFFFFF from word x.
3. Write 8F0F0F0F to word x.
4. Read 8F0F0F0F from word x.
5. Write 00000000 to word x.
6. Read 00000000 from word x.
```

```
7. Write 70F0F0F0 to word x.
8. Read 70F0F0F0 from word x.
9. Repeat for each word, x (x = 31 down to 0).
```

H.3 ESAM Word Select Circuit

Eight separate tests were created for the Word Select Circuit. The first seven tests were designed to validate the functionality of the circuit. These tests included the validation of functions that are not used in the NEQ Component but are available in the circuit. The eighth test was designed to measure the timing performance of the required functions within the NEQ Component.

H.3.1 Validation. The following tests were used for validation of the Word Select Circuit. These tests revealed that the correct `CTRL(3)` input for the fabricated circuit is the inverse of the `CTRL(3)` input for the VHDL model (as listed in Tables 23 and 24).

```
Test 1: Write-All.

Test 2: 1. Search-All (match on all).
        2. Write (read) one word at a time for all 32 words.

Test 3: 1. Search-All (match on subset).
        2. Write (read) one word at a time for all matches.

Test 4: 1. Search-All (match on subset).
        2. Write subset.

Test 5: 1. Search-All (match on subset).
        2. Search-Subset (match on smaller subset).
        3. Write subset.

Test 6: 1. Search-All (match on all).
        2. Search-Subset (match on all).
        3. Write subset (all words).

Test 7: 1. Search-All-Not (match on subset).
        2. Search-Subset (match on smaller subset).
        3. Write subset.
```

H.3.2 Performance. The critical path of the Word Select Circuit was the assertion of the search select outputs during a Search-All operation. The following test was used for the performance measurement of the Word Select Circuit:

```
1. Write-All.
2. Search-All (match on all).
3. Search-Subset (match on all).
4. Write (read) one word at a time for all 32 words.
5. Goto 1.
```

By running the test in an infinite loop, the polling time for the expected results (word selects and search selects) could be lowered until errors were detected. The minimum polling time at which errors were not detected was the minimum execution time of the circuit.

H.4 NEQ Control Unit

Only one test was created for the NEQ Control Unit. This test was designed to validate the performance of the circuit while simultaneously measuring the maximum operating frequency. The test was executed as follows:

```
1.  Reset.
2.  Initialize ESAM.
3.  Reserve arc (memory full error).
4.  Clear error.
5.  Initialize ESAM.
6.  Reserve arc.
7.  Reserve arc.
8.  Write to reserved word.
9.  Write to reserved word.
10. Write to unreserved word.
11. Write to unreserved word.
12. Write to unreserved word (memory full error).
13. Clear error.
14. Find min.
15. Find min.
16. Find min (event not ready error).
17. Clear Error.
18. Output Data.
19. Output address.
```

```
20. Search.
21. Search (no-valid-event error).
22. Clear error.
23. Goto 1.
```

This test provided functional validation of the circuit since all states were entered and the correct control outputs were observed. By running the test in an infinite loop, the polling time for the expected results (control outputs) could be lowered until errors were detected. The minimum polling time at which errors were not detected was the minimum execution time of the circuit.

H.5 Corrected ESAM Array

The corrected ESAM array was a 4-word by 6-bit memory which had improved write and read circuitry. The stuck-at fault on the comparand data path was also corrected so that equivalence search operations could be executed. Three tests were designed for the corrected ESAM Array. These tests were designed to:

- Validate the execution of associative and I/O operations.
- Measure the cycle speed for I/O operations.
- Measure the worst-case timing performance of the associative operations.

H.5.1 Validation. The following test was used to validate the execution of both the associative and I/O operations in the corrected ESAM Array. All vectors are binary.

```
1.  Write 111111 to all words (3 down to 0).
2.  Write 000000 to word 3.
3.  Search-All minimum (match on word 3).
4.  Search-All maximum (match on words 2 down to 0).
5.  Search-All equal to 000000 (match on word 3).
6.  Search-All equal to 111111 (match on words 2 down to 0).
7.  Read word 3 (output 000000).
8.  Read word 2 (output 111111).
```

This test demonstrated the improvements over the initial ESAM Array. The read operation was no longer stuck at a high logic value, and the equivalence search operation returned correct results.

H.5.2 Performance of I/O Operations. The following test was used to measure the cycle speed of I/O operations:

```
1.  Write 111111 to word x.
2.  Read 111111 from word x.
3.  Write 101010 to word x.
4.  Read 101010 from word x.
5.  Write 010101 to word x.
6.  Read 010101 from word x.
7.  Write 000000 to word x.
8.  Read 000000 from word x.
9.  Repeat for each word, x (x = 3 down to 0).
10. Goto 1.
```

Both the test cycle time and polling time for expected results (data output) were lowered until errors were generated. The minimum cycle time and polling time at which errors were not detected was the minimum execution time of the circuit.

H.5.3 Worst-Case Performance of Associative Operations. The following test was used to measure the worst-case timing performance of the associative operations:

```
1.  Write 111111 to all words (3 down to 0).
2.  Write 000000 to word 2.
3.  Search-All minimum (match on word 2).
4.  Write 000000 to all words.
5.  Write 111111 to word 1.
6.  Search-All maximum (match on word 1).
7.  Write 000000 to all words.
8.  Search-All equal to 000000 (match on all words).
9.  Write 111111 to all words.
10. Search-All equal to 111111 (match on all words).
11. Goto 1.
```

This test is similar to the test that was used to measure the performance of the associative operations of the initial ESAM Array. The primary difference is that this test also measures the performance of the equivalence search. The polling time for the expected result (match) was lowered until errors were detected. The minimum polling time at which errors were not detected was the minimum execution time of the search operation. The

Search-All minimum typically failed before the Search-All maximum, and the Search-All equal was always the fastest.

When one of the searches (max, min, or equal) began to fail, the expected result from the failing search was no longer polled. This technique allowed the polling time to be decreased until the faster searches began to generate errors. In this manner, the same test was used to measure the performance of all of the associative operations.

The above test was executed across search fields of 6, 5, 4, 3, 2, and 1 bit(s). The size of the search field was controlled by setting the mask input to the desired value.

Appendix I. DES Coprocessor Project Directory

This appendix explains the hierarchy of the DES Coprocessor project directory which is located on the AFIT VLSI network file server. "README" files are placed throughout the directory to further explain the contents of files.

```
\des - This is the root of the project directory.
   \esam - Contains all of the VHDL, Magic, and HSPICE files for
           the ESAM and NEQ Component.

           \extrema_array_fab\extrema_array_fab2 - Contains the Magic and
           HSPICE files for the 32-word by 32-bit ESAM array that was
           fabricated in a 132-pin PGA.

           \extrema_array_fab\extrema_array_fab3 - Contains the Magic and
           HSPICE files for the corrected ESAM Array that was fabricated
           in a TinyChip.

           \wd_sel_fab\wd_sel_fab2 - Contains the Magic and HSPICE files
           for the ESAM Word Select Circuit that was fabricated in a
           132-pin PGA.

           \vhdl - Contains the complete VHDL description of the NEQ Component.

   \ims - Contains all of the IMS test files for the fabricated circuits.

          \extrema_array_big - Contains the IMS test files for the
          132-pin PGA ESAM Array.

          \extrema_array_small - Contains the IMS test files for the
          TinyChip corrected ESAM Array.

          \neq_ctrl - Contains the test files for the TinyChip NEQ Control Unit.

          \word_sel_ckt - Contains the test files for the 132-pin PGA
          ESAM Word Select Circuit.

   \synthesis - Contains all of the synthesized chips that were fabricated.

          \des_execution_unit\layout - Contains the Magic layout of the
          coprocessor execution unit that was fabricated in an 84-pin PGA.

          \des_execution_unit\layout\esim_test - Contains all of the ESIM
          switch-level simulations that were run on the execution unit.
```

\neq_control_unit\layout - Contains the Magic layout of the NEQ Control Unit that was fabricated in a TinyChip.

\ver5 - Contains a predominantly behavioral description of the DES Coprocessor.

\ver6 - Contains a predominantly structural description of the DES Coprocessor.

For information on obtaining a copy of the DES Coprocessor files, send correspondence or call LtCol William Hobart at:

Air Force Institute of Technology
AFIT/ENG
Wright–Patterson AFB, OH 45433-7765
com. (513) 255-3636 ext. 4622
av. 785-3636 ext. 4622

Bibliography

1. Banton, David W. *A Decision Criteria to Select an Associative-Memory Organization that Minimizes the Execution Time of a Mix of Associative Search Operations*. PhD dissertation, AFIT/DS/ENG/93-02, Air Force Institute of Technology (AU), June 1993.
2. Breeden, Thomas A. *Parallel Simulation of Structural Circuits on Intel Hypercubes*. MS thesis, AFIT/GCE/ENG/92D-01, Air Force Institute of Technology (AU), 1992.
3. Buzzell, Calvin A. and others. "Modular VME Rollback Hardware for Time Warp." *Proceedings of the SCS Multiconference on Distributed Simulation*. 153–156. IEEE, 1990.
4. Chandy, K. M. and J. Misra. "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, *24*:198–206 (April 1981).
5. Chase, Paul W. and others. "Visualization of Parallel Discrete Event Simulations." Unpublished Article, September 1993.
6. Daniel, David W. *Design of a Hardware Discrete Event Simulation Coprocessor*. MS thesis, AFIT/GCE/ENG/93M-01, Air Force Institute of Technology (AU), 1992.
7. Franklin, M.A. and others. "Parallel Machines and Algorithms for Discrete-Event Simulation." *International Conference on Parallel Processing*. 449–458. Columbus, OH: IEEE, 1984.
8. Fujimoto, Richard M. "Parallel Discrete Event Simulation." *1989 Winter Simulation Conference*. 19–28. IEEE, 1989.
9. Fujimoto, Richard M. and others. "Design and Performance of Special Purpose Hardware for Time Warp." *International Symposium on Computer Architecture*. 401–408. IEEE, 1988.
10. Fujimoto, Richard M. and others. "Design and Evaluation of the Rollback Chip: Special Purpose Hardware for Time Warp," *IEEE Transactions on Computers*, *41*:68–82 (January 1992).
11. Gustafson, John L. "Reevaluating Amdahl's Law," *Communications of the ACM*, *31*:532–533 (May 1988).
12. Intel Corporation. *Microprocessors, Volume II*, 1991.
13. Jefferson, David. "Virtual Time," *ACM Transactions on Programming Languages and Systems*, 7:404–425 (July 1985).
14. Jefferson, David. "Virtual Time II: Storage Management in Distributed Simulation." Unpublished Article, December 1989.
15. Jefferson, David and others. "Distributed Simulation and the Time Warp Operating System," *ACM Operating Systems Review*, 77–93 (November 1987).
16. Kapp, Kevin L. *Partitioning Structural VHDL Circuits for Parallel Execution on Hypercubes*. MS thesis, AFIT/GCE/ENG/93D-14, Air Force Institute of Technology (AU), 1993.

17. Misra, Jayadev. "Distributed Discrete-Event Simulation," *ACM Computing Surveys*, *18*:39–65 (March 1986).

18. Motorola, Inc. *MC68EC030 32-Bit Embedded Controller User's Manual*, 1990.

19. Reed, Daniel A. and Allen D. Malony. "Parallel Discrete Event Simulation: The Chandy-Misra Approach." *Distributed Simulation*. 8–13. La Jolla CA: SCS, 1988.

20. Reynolds, Jr. Paul F. "A Spectrum of Options for Parallel Simulation." *1988 Winter Simulation Conference*. 325–332. IEEE, 1988.

21. Reynolds, Jr. Paul F. "Comparative Analyses of Parallel Simulation Protocols." *1989 Winter Simulation Conference*. 671–679. IEEE, 1989.

22. Reynolds, Paul F. and Carmen M. Pancerella. *Hardware Support for Parallel Discrete Event Simulations*. Technical Report TR-92-08, University of Virginia, 1992.

23. SanGregory, Sam L. *A Single-Chip 2K x 8-Bit Pipelined Digital RF Memory Using 2μm CMOS VLSI Technology*. MS thesis, AFIT/GCE/ENG/92D-10, Air Force Institute of Technology (AU), 1992.

24. Taylor, Paul J. *Requirements Analysis for a Hardware, Discrete-Event Simulation Engine Accelerator*. MS thesis, AFIT/GCE/ENG/91D-11, Air Force Institute of Technology (AU), 1991.

25. Xilinx, Inc. *The XC4000 Data Book*, 1991.

Vita

Captain Berlin was born in New Orleans, Louisiana on 29 April 1963. He graduated from De La Salle High School in 1981. He attended the United States Military Academy and graduated with a Bachelor of Science Degree in 1985. He was commissioned in the Corps of Engineers and was assigned to the 16th Engineer Battalion, 1st Armored Division. He was next assigned to the 7th Special Forces Group, Fort Bragg, NC. In 1992 Captain Berlin was selected to attend the Air Force Institute of Technology for completion of a Master of Science in Computer Engineering.

Permanent address: 5237 Berkley Drive
New Orleans, Louisiana 70114

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1993	Master's Thesis

4. TITLE AND SUBTITLE
DESIGN OF A PARALLEL DISCRETE EVENT SIMULATION COPROCESSOR

5. FUNDING NUMBERS

6. AUTHOR(S)
Jacob L. Berlin

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/93D-02

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
ARPA/CSTO
Dr Robert Parker
3701 N. Fairfax Dr
Arlington, VA 22203

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

A Parallel Discrete Event Simulation Coprocessor was designed to off-load the synchronization overhead from the processors executing the application. In a multiprocessor architecture, one coprocessor executes the synchronization routines for each host processor. Speedup can be achieved when the host processor executes the application and the coprocessor concurrently executes synchronization routines. The coprocessor uses a programmable microcode control store to guarantee flexibility in the synchronization routines.

The coprocessor uses an Extreme Search Associative Memory to support fast Next Event Queue (NEQ) management. This associative memory uses bit-serial word-parallel search logic to provide O(1) insert and retrieval time of events in the NEQ.

The coprocessor was completely described in the VHSIC Hardware Description Language (VHDL), and several components were fabricated and tested. Timing measurements of the fabricated components were back-annotated into the VHDL description to improve model accuracy.

Synchronization overhead of a parallel VHDL simulation was measured using the AFIT Algorithm Animation Research Facility, and this data was used for a conceptual performance analysis of the coprocessor. A four-fold speedup was achieved for the NEQ management of the simulation; however, the total speedup was only 1.02 since less than 2% of the application was accelerated.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Parallel Discrete Event Simulation, Parallel Architecture, VLSI Architecture, Associative Memory	xxx
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL